ROBUST AND EFFICIENT MALWARE ANALYSIS AND HOST-BASED MONITORING

A Thesis
Presented to
The Academic Faculty

by

Monirul I. Sharif

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2010

ROBUST AND EFFICIENT MALWARE ANALYSIS AND HOST-BASED MONITORING

Approved by:

Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

Jonathon Giffin, Co-advisor
School of Computer Science
Georgia Institute of Technology

Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

Santosh Pande
School of Computer Science
Georgia Institute of Technology

Douglas Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: September 2010

To my parents,

and to those who dedicate their lives

for the betterment of others,

expecting little to nothing in return.

ACKNOWLEDGEMENTS

It is a pleasure to thank those without whose help and support this thesis would not have been possible. I owe my deepest gratitude to my advisor Wenke Lee for his uncomparable guidance and support over the years. Because of his thirst for excellence, steady patience and belief in my abilities, I was able to have enough freedom to choose and focus on challenging yet impactful problems that I felt deeply passionate towards. I am grateful to my co-advisor, Jonathon Giffin, who helped me build several skills necessary to be a good researcher and with whom I had the pleasure of working on several interesting research problems. I would love to thank my other committee members Mustaque Ahamad, Santosh Pande and Douglas Blough for their valuable comments and support regarding the thesis and my research. I would like to express my gratitude and best regards to Weidong Cui, whose support, collaboration and guidance was essential for my research and the thesis.

I am indebted to many of my colleagues and friends of my research group at Georgia Tech. It was my pleasure to be able to work with my friend and colleague Andrea Lanzi, whose time here at Georgia Tech resulted in a period of great teamwork and collaboration on many interesting projects. I would love to thank Paul Royal whom I had the pleasure to work with on many problems. I had the opportunity to colloborate with many of my friends in the group, including Kapil Singh, Bryane Payne and Martim Carbone. Besides them, I am thankful to all the other friends in my research group for their support and for providing a wonderful environment that made work and research an enjoyment.

I would not have succeeded in finishing my PhD in the way I have without the support from my community and friends. Among many, I would like to specially

thank Nova, Arshad, Moin, Farzana, Lopa, Shajib, Abir, Towhid, and Shefaet. My life in Atlanta was always full of joy and happiness for all of my friends.

I am deeply grateful to my family for their love and support. Without any doubt, the unwavering and unconditional love and inspiration of my parents throughout my life has lead me to be were I am. My father, who is a constant source of inspiration and confidence, was the role model I followed throughout the life. My mother's affection and view of life has made me see the world from a different perspective and has given me the courage to go through the struggling periods of my PhD. Words are not sufficient to express how grateful I always am to my brother Roni and my sister Lipa. Even though they were far away during my PhD, I would feel them always close, being there for me at the moments I would need them the most.

I would like to thank my wife, Farhana Aleen, whose support, inspiration and love was absolutely essential for success of the work done for PhD dissertation. At times, our common big dreams in life would sometimes help fight and get through the most challenging moments in my PhD.

Above all, I thank the One Whom I am ever grateful for bestowing guidance, intellect, knowledge, patience, wisdom, and essentially everything virtuous that might exist in me. Lastly, I offer my regards and blessings to all of those who I have not mentioned, but who have supported me in any respect during my PhD student life, which lead me to produce this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Over the last few years, a tremendous increase has occurred in the rate in which new malware is appearing in the Internet. Today, organized cybercriminals are using malware as their primary vehicle for carrying out various cyberattacks on computers for huge financial gains. On the defense side, host-based malware detection approaches such as antivirus programs are severely lagging. Industrial average detection rates range from 18% for zero day to 60% for one month old malware samples. The two important aspects that the overall effectiveness of malware detection depend on are the success of extracting information from malware using malware analysis to generate signatures, and then the success of utilizing these signatures on target hosts with appropriate system monitoring techniques. Today's malware employ a vast array of anti-analysis and anti-monitoring techniques to deter analysis and to neutralize antivirus programs, reducing the overall success of malware detection.

In this dissertation, we present a set of practical approaches of *robust* and *efficient* malware analysis and system monitoring that can help make malware detection on hosts become more effective. The contributions are summarized below:

(1) Efficient Methods for Enabling Static Malware Analysis: Static malware analysis suffers greatly for its susceptibility to obfuscations employed by malware. However, it can provide complementary insight to dynamic analysis in those cases where obfuscations can be sufficiently overcome. We present Eureka, a framework that efficiently deobfuscates single-pass and multi-pass packed binaries and restores obfuscated API calls, providing a basis for extracting comprehensive information from the malware using further static analysis.

(2) Making Dynamic Analysis Approaches more Robust: While dynamic analysis

techniques provide better resilience to malware obfuscations than static analysis, it is susceptible to evasion attacks that detect the presence of a run-time analysis environment. We present the formal framework of *transparent malware analysis* and *Ether*, a dynamic malware analysis environment based on this framework that provides transparent fine-(single instruction) and coarse-(system call) granularity tracing.

(3) Anticipating Obfuscations that Hide Trigger-based Behavior: Multipath exploring dynamic analysis overcomes the limitation of straightforward dynamic analysis, which may miss trigger-based behavior for its limited view of execution paths. We introduce an input-based obfuscation technique that hides trigger-based behavior from any input-oblivious analyzer. We present the analysis of strengths and weaknesses of this obfuscation and explain how such a technique can impact the efficiency and effectiveness of malware analysis.

(4) Reversing Emulator based Obfuscation: Recently, malware authors have adopted emulation as a forms of fine-grained obfuscation that can affect the robustness of both white-box and gray-box analysis techniques. We present an approach that automatically reverse-engineers the emulator and extracts the syntax and semantics of the bytecode language, which helps constructing control-flow graphs of the bytecode program and enables further analysis on the malicious code.

(5) Robust and Efficient System Monitoring Techniques: Antivirus programs require monitoring of the target host for code and events that are matched with the signatures or behavior models that they employ, overcoming disabling attacks used by malware. We present Secure In-VM Monitoring, an approach of efficiently monitoring a target host while being robust against unknown malware that may attempt to neutralize security tools.

CHAPTER I

INTRODUCTION

1.1 Motivation and Goals

As computers and the Internet are becoming a core necessity for individuals and businesses throughout the world, cybercriminals are continuously finding new opportunities for breaching computer security for huge financial gains. Malicious programs or scripts, or in general *malware*, are now used as platforms for launching cyberattacks on computers to realize these goals. From the viruses that were once written to show off competency and technical prowess, malware has evolved and taken into many forms, including worms, bots, spyware, adware, rootkits, etc., each with a different approach of exploiting users and business systems to make money for the now very organized underworld. Recent events show that the masses are not the only targets. Skilled malware authors are even making targeted attacks at organizations for goals such as corporate espionage [170, 8, 121]. Conservative estimates [12] show reported annual losses in around US $14 billion per year in the United States alone due to cyberattacks from Malware. It is without any doubt that we are in an era when the need for defense against malware is more than ever before.

Since any action and event caused by running programs can be visible at the host level, host-based security tools are being used as the primary method of defense against malware. Among all host-based security approaches, *Malware detection* is still seen as the most viable and practical approach, and tools such as antivirus programs are software are used as the primary defense against malware. At present, the vast majority of host-based security tools, including *anti-virus* software, still use *signature-based* detection approaches. A malware is successfully detected on an end-host if the

code, behavior or a combination of actions observed by the tool match a specific signature in the signature database. Even behavior based detection require updates to the behavior model that is used for detection [148].

Over the last few years, security tools and antivirus software continue to lag in terms of their malware detection capabilities. The current average detection rate for all leading antiviruses starts at 18% for zero-day and goes upto 60% after 30 days [2]. With tens and thousands of new malware samples appearing everyday [13], antivirus companies are taking long period to have signatures available as updates for security tools that are deployed. This means that a large fraction of malware can perform their malicious activities undetected on infected systems.

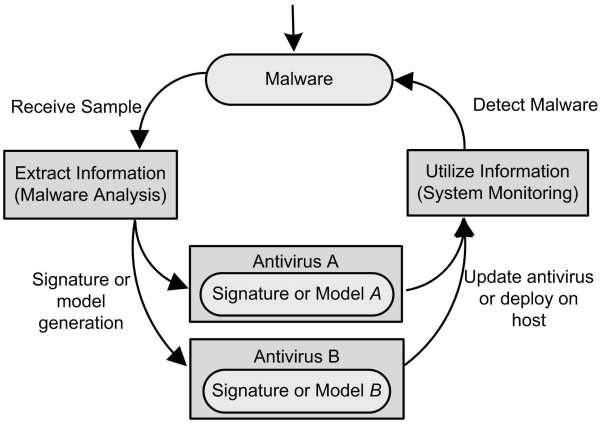


Figure 1: The malware detection process.

While advances can be made in research on how malware code or behavior can be more accurately represented by signatures or models, there are opportunities for improving how accurately information is *extracted* from malware to generate signatures and how the signatures are *utilized* by antivirus programs to detect malware. By improving the effectiveness of both of these aspects, the overall effectiveness of malware detection can be improved in general. This thesis explores such opportunities.

1.1.1 Malware Detection Process

We call the entire process starting from the time a malware instance is received by an antivirus company to the time when the malware is successfully detected on a malware on an end-host as the *malware detection process*. Figure 1 shows a general view of this process. There are three important aspects of the process that determine the the success of detection.

First, when the malware sample is received, a malware signature is generated by using information *extracted* from the malware through a very complex task of *malware analysis*. The analysis is usually a method of *reverse engineering* the malware with a goal of understanding what it does and how, or extracting useful code or behavior that can used to identify the malware. This process involves defeating various defenses and protections that malware employ to prevent or impede analysis. Its effectiveness is determined by how accurately information required for the signature can be extracted from the malware.

The second aspect is how the signature is *utilized* by the antivirus program while running on the host that is intended to be protected. The process, which is a form of *system monitoring* applied to the host for intercepting events or code that can give information that is matched with the signature. The effectiveness of this process depends on how accurately the the antivirus program can intercept events occurring on the target host, and how well the antivirus program is protected.

The third aspect is that the formation of the signatures themselves greatly affects the malware detection approach. The signature's effectiveness depends on how accurately it can capture the characteristics of the malware instance along with its derivatives or variants. While in the past simple static code sequences were sufficient, today, complex semantics and behavioral signatures [43, 79] are used to represent the properties of the malware.

In order to improve malware detection, research is pushing on identifying various problems and challenges in all these three fronts. However, attacks on malware analysis and system monitoring techniques can render malware detection unsuccessful even if highly accurate signature schemes are invented over the years. Therefore, identifying limitations in malware analysis and system monitoring techniques and improving upon them in general can benefit malware detection irrespective of the signature generation or modeling schemes.

1.1.2 Limitations of Current Malware Analysis Approaches

Unfortunately, malware developers are well aware of the efforts to reverse engineer their binaries, and employ a wide range of binary obfuscation and evasion techniques to deter analysis and reverse engineering. Therefore, there is a huge challenge in overcoming these obstacles and analyzing malware. Now multiply this challenge by the millions of new strains and repurposed malware variants that appear on the Internet monthly [94, 105], and the need to develop *automated tools* to extract and analyze all facets of malware binary logic becomes clear. Based on the methods of analyzing programs, automated malware analysis can be generally divided into three classes, each of which has their advantages and disadvantages.

Static Analysis or White-box Approaches: Historically, the first malware analysis approaches used static analysis, which attempts to analyze the code without executing it. Static analysis has a well-investigated background in research and is attractive because of being amenable to a vast range of useful set of techniques that can relate parts of code from the entire visible program to form higher level understanding of semantics and behavior. However, malware authors exploit the fact that static program analysis has to deal with many undecidable problems such as pointer analysis, and that accurately disassembling x86 binary code is hard due to the variable length **x86** instructions. Moreover, various code obfuscations that have

appeared over the years, directly using static on malware today is little to no use at all. Nevertheless, many signature schemes that rely on semantics [41] or higher-level behavior, may benefit from static analysis.

Pure Dynamic Analysis or Black-box Approaches: Since these approaches directly execute the malware and observe the external actions, any obfuscations targeted towards impeding static analysis are overcome. However, one of the main drawbacks of these approaches is that during execution of the malware, only the effects of a single execution path may be seen. In other words, any *trigger-based behavior* may be missed. These approaches are useful if external activities, such as system calls, network communication, changes in the file-system, or modifications to the OS environment need to be analyzed. Virtual machines are widely used as the analysis environments for these black-box based approaches. A large fraction of malware today employ various run-time checks to detect artifacts in the execution environment that indicate well-known analyzers or virtual machines.

Hybrid or Grey-box Approaches: The malware analysis approaches that combine static and dynamic analysis fall into this category. These analysis approaches have the ability to analyze the malware program code while executing it. Most grey-box approaches observe the execution of the malware at a fine-grained instruction level, providing a way to construct the program code, follow data-flow or even build a view amenable to static analysis for the executed path. Research has also enabled executing multiple paths in dynamic analysis, providing most of the benefits of static analysis without having its drawbacks. However, instruction-level execution introduces high overhead and also provides many side-effects that can be used by the malware to detect the analysis environment and avoid analysis.

Each method has its distinct advantages, and a malware signature generation approach will pick one of these approaches depending on the information required from the malware. However, if malware authors focus on exploiting weaknesses in

order to impede an analysis process, the malware detection methods that are based on the analysis can be severely impacted. Therefore, it is imperative that the weaknesses of these approaches are investigated and focus is placed on building techniques that can overcome possible threats.

1.1.3 Limitations in System Monitoring Techniques

Antivirus programs of today reside in the same operating system environment as it is protecting. In order to protect the antivirus code and data from malware that can infect the system, the antivirus program has to be isolated and the probes or hooks need to be protected. Moreover, the antivirus program should be able to intercept the events in the host that are being used in the signatures. Although Kernel based antivirus modules are isolated from user-level malware, the existence of rootkits make this approach does not guarantee isolation from the malware. Moreover, probes placed by the antivirus program need to be protected with higher privilege. Many host-based security systems use virtual machines to achieve *robustness* by keeping security tools in a separate OS environment and use introspection to read data from the target host. While this passive nature in inspection is not sufficient for most signatures used for malware detection, recent approaches for active monitoring [110] can intercept various events in the target host from out side has also been proposed. However, such a system loses the *efficiency* achievable by making the security tool reside in the same host, which in an affect may not allow the same events to be monitored as required by the signature. Both efficiency and robustness need to be achieved to improve the state of anti-malware.

1.2 Thesis Overview

This thesis explores **a set of practical approaches of robust and efficient malware analysis and system monitoring in order to improve malware detection on hosts**. Given various strategies for automated malware analysis, the first

goal that is focused on in this work is to make analysis more *robust*. This is vital for accurately extracting information about a malware for use in signatures by defeating as many anti-analysis tricks as possible employed to thwart analysis. By making malware analysis more *efficient*, the overall time required to produce signatures can be reduced, improving the effectiveness of detection by releasing signature updates as early as possible. *Robust* and *Efficient* system monitoring approaches can protect the antivirus program and its probes, ensuring that the known malware are properly detected and the unknown malware cannot neutralize the program. The techniques presented in the thesis are orthogonal to the signature-generation or behavior model based approaches used for detection. Therefore, the likelihood that a malware will be successfully detected by any signature-based malware detection approach is increased. The thesis first focuses on research problems in malware analysis and then moves onto improving system monitoring.

Chapter 2 presents background and related works in the area of malware detection, malware analysis, and system monitoring. The chapter contains a classification of various anti-analysis and anti-monitoring defenses that malware employ or research has investigated and presented. The thesis then presents five research topics on malware analysis and system monitoring through chapters 3 to 7, and the contents are highlighted below:

Chapter 3 investigates efficient methods to enable static malware analysis. Although a large number of obfuscations can impede static analysis, we show how the classes of single pass and multi-pass code encryption or packing obfuscation can be efficiently thwarted to reveal code suitable for performing successful static analysis. We also present how common attacks that try to hide system call invocation points from within the code can be reconstructed with a method which we call API resolution. The motivation behind this work is to allow efficient processing of malware samples, so that easy obfuscations can be defeated quickly. The work is based on

Eureka, which we have presented in [131].

We investigate in Chapter 4 how dynamic analysis can be made robust against evasion attacks performed by malware that detect the analysis environment and perform something unusual. We present the foundations of *transparent malware analysis* that is applicable both for pure dynamic and grey box approaches. The formal requirements of such a framework is first identified, and then summary of a system based on this approach called *Ether* [51] is presented.

While most obfuscations that encrypt code have the key inside the code, we present a new attack in Chapter 5 called the *Conditional Code Obfuscation* [127], which receives the decryption key from the input. This obfuscation technique can hide any trigger dependent code from all classes of malware analysis, including hybrid analysis approaches. However, the analysis cannot have any prior information about the possible inputs to the malware. The research work shows how such an attack can be carried out on existing code automatically using a compiler framework. Possible weaknesses and methods to tackle such an obfuscation is then discussed.

In Chapter 6, an obfuscation attack that uses emulation to hide the original code of malware is discussed. This obfuscation uses a and an automated way to reverse-engineer such an obfuscation is presented [128]. The attack is applicable to all types of malware analysis, including the greybox methods. However, we show that our solution is able automatically determine the syntax and semantics of the bytecode language and generate CFGs of the bytecode.

Finally, In Chapter 7, we present a method that makes secure monitoring robust and efficient enough that it can be used to protect a security tool that resides in the same OS environment of the target host, but it is secure enough as placing in isolated separate virtual machine. This technique, which is called *Secure In-VM Monitoring* [130], combines the benefits of out-of-VM monitoring with the efficiency of save VM efficiency.

We conclude the thesis and present a few open-ended problems in Chapter 8.

CHAPTER II

BACKGROUND AND RELATED WORK

2.1 Malware and Evolution of Their Defenses

It has been several decades since the first malware was written [161]. Over the years, the intention and motivation behind writing malware have changed immensely. The first generation of malware, which was in the form of computer viruses, was primarily written with the motivation of showing off skills, and in many cases sheer self-satisfaction. The payloads in the viruses were mostly informative, showing a system has been infected, and in some cases destructive in nature [159], resulting in loss of data or BIOS. Moreover, the method in which viruses would need to propagate or multiply had the side-effect of modifying valuable programs or user data. Even then, malware authors found it important to keep their creations alive and working as long as possible. In order to defeat detection systems or human analysis used encryption [169], polymorphism [139], and various other obfuscation schemes [33]. The security tools and antivirus programs continued to evolve in order to be more resilient, more adaptive and more accurate in detecting malware [143].

The biggest shift in the way malware is written came when the malware authors found a new motivating factor behind creating and deploying malware. Driven by the powerful motivating force of monetary gains, malware authors became highly organized and several intuitive forms of malware started to appear with the intention of violating security and exploit the modern methods of electronic commerce. Large worm outbreaks [19, 99, 98] showed that malware can reach thousands to millions of systems worldwide in a small amount of time. The `ILOVEYOU` worm was reported

to reach almost 50 million infections and caused an estimated $5.5 billion in damages worldwide. With the advent of *bots*, malware authors can have vast armies of computers at their disposal for achieving numerous malicious objectives. Similarly, malware appears in the form of spyware, adware, rootkits, etc., with different intuitive malicious goals.

The number of malware programs released every year has grown exponentially. An antivirus company [13] has reported that in 2008, the total number of new malware released is almost equal to that of the previous 17 years. The same source has also reported more than 22,000 new samples being received daily during that year. We are now amidst an era of malware where defense is mostly playing catchup and researchers and the industry is looking for automated ways to tackle the thousands of malware programs appearing everyday. On one hand, as researchers and security industry fight to improve malware defense, the malware authors continuously introduce new ways to attack offline analysis and run-time checks to evade detection.

2.1.1 Earlier Attacks on Malware Detection

One of the earliest attacks on malware detection were to deter signature based schemes. Signatures of malware were created by taking a sequence of characters from the body of the malware, which were mostly revealed using static analysis. In order to defeat these signatures, *oligomorphic* viruses [143] used to encrypt or encode the body of the virus with a random key during every infection and attach a small decryption code that decrypts the code at run-time. *Polymorphic* viruses take this one step further and have varying decryption code to make finding signatures harder. However, since all the encrypted code needs to be decrypted before execution, anti-malware tools employed dynamic analysis or emulation to help extract the code. *Metamorphic* viruses addressed this issue by morphing the code itself with various

tricks like register renaming, code reordering, equivalent code replacement, etc. However, metamorphic engine requires a compiler to be carried with a malware, making them highly error prone and unrealistic. These earlier principles have evolved into more recent code packing techniques used by packers [108, 132] that we discuss later.

In order to identify possible signatures, human experts used a combination of run-time execution and debuggers in an effort to overcome the code encryptions. Malware programs started to detect debuggers' [143] presence by looking for breakpoints, looking for debugging API usage and existence of processes or drivers related to debuggers. Once presence of these tools are detected, malware can evade analysis by changing the behavior or terminating execution. While these tools have been replaced by automated analysis tools recently, the fundamental goal of newer evolved attacks of detecting analyzers at run-time remain the same.

2.1.2 Recent Forms of Attacks

In order to reduce the time required to introduce new malware, malware authors started using commercially available software protectors or packers as the method to protect their programs [132]. These tools can take a compiled binary and then add several layers of protections on the malware to make malware detection hard. The commercial tools mostly attempt to make analysis of the protected programs hard. However, malware authors add additional defenses against detection that is specific to anti-malware and antiviruses. The attacks against detection can be classified into *anti-analysis* and *anti-monitoring* categories where they target malware analysis and system monitoring, respectively. The anti-analysis techniques can also be divided into *code obfuscation* and *analysis evasion* techniques depending on whether they target static or run-time analysis.

2.1.3 Code Obfuscation

A large number of attacks fall in the category of *program obfuscation* that aims at making statically analyzing a program hard. The first group does not require any additional transformations at run-time to make program executable, but the modified program is executable code that is harder to analyze than the original. On the research side, several papers have been proposing possible techniques that can deter static analysis. [33] provide a very early compilation of various obfuscating program transformations that had been proposed till then. Since the variable length instructions of the x86 processor create a problem to differentiate between code and data, this can be utilized by various attacks to make disassembly inaccurate. For example, most recent disassemblers, use recursive descent methods to follow control-flow and identify more code accurately. Hiding control-flow is a proven technique to make identification of code harder. This is exploited in the [35] where disassembly is made inaccurate by inserting code while compilation. Opaque Predicates [34] is a technique that adds conditions to a program that is hard to statically solve. By using such predicates, the control-flow of a program can be hidden, making analysis inaccurate. Similar techniques have been proposed that use signals [116] to disrupt identifying the flow of execution.

Following the techniques of polymorphism and metamorphism in the early viruses, packers employ complete code encryption with a small decryptor that decrypts or decodes the code into executable form when executed. The simplest form can use transformations such as `XOR`, and it can go up to the complicacy of using encryption with a specific key. However, these packing techniques mostly include the obfuscation key as part of the program. Several unpacking techniques have been proposed that using dynamic analysis or run the malware to a certain extent to reveal the packed code [122, 76, 96].

Packing techniques can vary in terms of *packing granularity*. At the simplest form,

a packer can use *single-pass packing* where the entire malware program is encoded or encrypted using one key and during execution, unpacking takes place and the entire code is unpacked before execution starts.

Multi-level or *Multi-pass* packing techniques that are employed by some packers use several layers of encoding and encryption and requires several phases of decoding when executed. The simplest packers in this category apply multiple layers of single-pass encoding, which makes all layers of unpacking happen before actual malware code executes.

Page-level unpacking was employed by a few packers initially [132] which actually unpacks code at the granularity of pages during execution. Therefore, code is unpacked and revealed on demand and depending on the execution path of the program. Techniques such as this make it hard for unpackers to actually unpack the entire code because as execution proceeds, partial code can be revealed only for the paths that were executed. Therefore, making such packing not only appropriate for static analysis, but also dynamic analysis because only a single execution path may be revealed.

It can be observed that the smaller the granularity of packing, the smaller the amount of code gets revealed during execution. Which also affects how much code can be revealed by unpacking techniques. The finest granularity of obfuscation that is possible is at the *instruction-level*. It can be theoretically conceived that a packer may be able to unpack each instruction during execution time, but such a technique can be impractical for use, primarily due to the overhead incurred during unpacking. However, techniques similar to dynamic translation [135] can enable smaller granularity packing.

A technique that can be considered a practical comparable to instruction-level packing is *emulation-based packing*. Recent malware have started using emulators as a form of obfuscation by converting programs into random bytecode and then using

emulators to execute the bytecode on real hardware. Such a technique does not reveal the actual instructions during execution, but executes equivalent instructions in the emulator that execute the same semantics. In Chapter 6 of this dissertation, we present a new approach of defeating emulation based packing.

Although all these packing techniques use keys for encryption that remain in the code, we anticipate attacks that can be encrypted with keys that are not part of the program and come from inputs. We can call such techniques *input-based obfuscation*. In order to help improve analysis, we propose an input-based obfuscation technique that can hide trigger-based behavior. This attack, which is called *Conditional code obfuscation* is presented in Chapter 5.

2.1.4 Analysis Evasion

Since the malware is executed during dynamic analysis, it is possible to write code that during execution, performs some checks to detect an analyzer and evade analysis by showing random behavior or terminating execution. Earlier attacks that detect debuggers or system call tracers are now evolved to detect specific analyzer environments. For example, attacks detect QEMU [75, 95] because it is used by a wide range of fine-grained or greybox malware analysis techniques. Since virtual machines are also very useful for performing malware analysis, many malware employ checks to evade these environments by testing for virtual machines monitors such as VMWare, Virtual PC, etc.

2.2 Malware Analysis

Over the years, various malware analysis have evolved depending on the needs of malware detection techniques and also the continuous improvement of anti-reverse engineering methods utilized by malware authors. Although the fundamental traits of malware analysis come from the program analysis research, the fact that there exists an adversary that tries to defeat the analysis, makes the utilities and challenges vary

greatly.

There has been a number of techniques that have utilized static analysis for automating the identification of malware or generating signatures. Techniques for generating signatures using static analysis are presented in [40]. Since the approach is mostly dependent on patterns of code, it does not work for transformations such as metamorphism. Semantic-aware detection or signatures provided a technique of detecting malware based on semantics of the program rather than the syntax, making it amenable to a few classes metamorphic transformations. In [84], static analysis was used to detect rootkit like patterns in drivers. Finally, [79] uses static analysis to identify spyware by looking for behavior of leaking private information by browser helper objects (BHO).

Several automated malware analysis techniques use dynamic analysis [24, 45, 11, 146, 36]. Although the earlier systems used system-call tracing for the basis of analysis [36], the need finer-grained instruction-level analysis made emulators such as QEMU [26] an attractive run-time analysis platform. Recent versions of CWSandbox and VMScope [30] uses emulation to perform run-time dynamic analysis. for understanding malware behavior. Instruction level analysis techniques generally use emulators that allows monitoring of the effect of every executed instruction in a system and identify data modification patterns and at the same time protect the analyzer which resides outside o the malware execution environment. Methods such as dynamic tainting [103] or dynamic slicing requires instruction level tracing. Fine-grained instruction level analysis is especially important for understanding and detecting kernel level attacks. The main reason is that kernel level code does not use a fixed interface such as system calls like user-level code, and it can manipulate any data residing in the kernel.

HookFinder [167] performs fine-grained impact analysis using taint propagation to identify how a rootkit places hooks in the kernel execution path. HookMap [157]

identifies all potential hooking locations in the kernel execution paths. Both these systems help understand how known rootkits can modify legitimate kernel level behavior and introduce its malicious activities. While both HookFinder and HookMap focus on extracting control-flow modifications, K-Tracer [87] can identify both control-flow and data modifications that are related to a particular system call. K-Tracer uses a unique combination of forward and backward slicing techniques to precisely identify modifications to incoming arguments or outputs of a system call.

Besides kernel malware, fine-grained monitoring and analysis has used for a large number of user-level malware analysis approaches [168, 52, 122, 76]. Panorama [168] uses full system taint tracking to analyze privacy breaching malware. Unpackers such as Polyunpack [122], Renovo [76], etc use look for execution of instructions that are newly introduced by the malware to identify unpacked code.

Dynamic analysis techniques inherently overcome all anti-static analysis obfuscations, but they only observe a single execution path. Malware can exploit this limitation by employing trigger-based behaviors such as time-bombs, logic-bombs, bot-command inputs, and testing the presence of analyzers, to hide its intended behavior. The multipath exploring dynamic analysis technique [100] explores multiple paths in a program by saving the program state at a specific condition, continuing execution on one branch, restoring the state, solving the memory constraints for another branch and continuing execution along that branch.

2.3 Host-Monitoring and Anti-malware Security Tools

An essential component of an entire host-based malware detection system is the host-based security tool or antivirus that actually attempts to detect malware on a given host. While the antivirus programs or security tools have evolved over the years to counteract various sophisticated tricks that malware authors have introduced over the years, the general design has not been changed. The fact the anti-malware tool uses

a knowledge base (such as a signature database) that contains information that can be used to detect malware, leads to the requirement of gathering similar information from the host that is being protected in order to attempt to find a potential match. This process of gathering information is what can call as *host-based monitoring*.

In the earlier era of antivirus programs [160], antivirus programs where mostly designed to detect viruses. Since the first viruses infected executable files, these antivirus programs would scan the memory and files in the disk to detect the existence of viruses. The detection was performed by searching for matches with signatures, which were primarily a sequence of bytes in the viruses that uniquely identifies the virus. In these approaches, since the actions of the scanning is not required to coincide with any action of the virus, they can be considered as *passive* in nature.

As antivirus tools became more sophisticated and the need for detecting early to prevent widespread damage became substantial, the tools intercepted specific events in host and then used information gathered from these events or information available during the time of the event to detect malware. An example is checking for changes in a executable file every time it is opened, to check to see whether a virus has modified it. Similarly, antivirus programs now need to intercept several important events occurring in the host to detect malware, especially because behavior based heuristics [143] are being employed in some antiviruses. These monitoring of events can be called *active* because of their requirement of coinciding monitoring with a particular event occurring in the system.

Regardless of whether passive or active monitoring is used, the information viewed by the tools need to be trustworthy and the tools themselves need their functionality to be unaltered in order for detection to succeed. Earlier viruses that could manipulate the view of the system that an antivirus is seeing were called *Cobra* viruses [146]. These viruses that existed in the era of `MS-DOS`, would hook into software interrupts that would service application programs and modify the results or behavior of theses

services. MS-DOS was not designed to restrict such modifications by application programs to the system, allowing such modification to be done. Today, rootkits [68] achieve similar goals by installing themselves as kernel modules and changing system behavior to hide malicious activities. For such malware to succeed, they need to be at least at the same level of privilege as the antivirus programs in order to modify their probes inserted in the system, or modify their behavior to neutralize their detection capabilities. Research has moved into techniques for gaining higher privilege than the kernel using *virtualization* technology. First, earlier approaches that have the security tools reside in the host are discussed, and then then earlier work that use virtualization for securing and monitoring the host are presented.

2.3.1 Traditional Active Monitoring Inside the Host

A large volume of work that focuses on using monitoring but resides in the same operating system of the host is in the area of host-based intrusion detection. The goal of these security approaches are to protect application programs from external attacks. Therefore, they involve monitoring of events carried out by the applications. The monitoring of events vary in granularity, ranging from coarse-grained events such as system calls to fine-grained events such as control-flow. However, an important distinction between these approaches and malware detection is that they assume the malicious code attack comes in the form of an input to a legitimate application that needs to be protected, and that the operating system is trusted. For malware detection, the entire system needs to be assumed untrusted because it could be compromised by malware.

2.3.1.1 System Call based Monitors

Numerous prior systems detect application-level attacks by observing process execution at the granularity of system calls [54, 63, 64, 81, 89, 151, 40, 65, 60, 125]. These

approaches build a model of the legitimate behavior of an application and then monitor the application at run-time to detect anomalies. A malicious code attack that exploits a vulnerability in an application Rather than directly detecting the execution of malicious code, these tools attempt to detect attacks through the secondary effect of the malicious code or inputs upon the system call sequences executed by the process. By allowing attack code to execute, these secondary detectors provide attackers with opportunities to evade detection. Mimicry attacks [152, 144, 66] succeed by appearing normal to a system-call sequence based detector. System call models have grown in complexity to address mimicry attacks, but remain vulnerable because they allow invalid control flows to execute [83].

The general approach of these systems differ from malware detection, which primarily looks for malicious behavior instead of looking for deviations in legitimate behavior.

2.3.1.2 Inlined Reference Monitoring

IRM has been widely adopted as a faster method of ensuring safety properties in programs by including the monitor inside the program it is monitoring. This is a far more efficient way than to have a reference monitor that resides in the kernel for user-space programs, or in the hypervisor for kernel-space programs. Traditional approaches of ensuring the integrity of the monitor itself for IRM techniques has been to ensure specific data-flow and control-flow safety properties throughout the program. For example, SFI [153] (Software Fault Isolation) is a method for having untrusted program share the same address space and provide isolation. This is achieved by rewriting specific store operations at compile time so that the address is masked in a way that it cannot write to an address region. SFI is well suited for application programs that can be modified during compile time. Achieving it for the kernel, which might have varying policies, is a hard problem.

Control Flow Integrity (CFI) [14] instruments control-flow instructions with checks and their possible targets with labels at compile-time so that at run-time the checks enforce control-flow to be in the static CFG of the program. Since CFI covers all control-flow instructions, it also prevents circumvention of any of its checks. XFI [53] is an extensible fault isolation framework that provides fine-grained byte level memory access control. These features along with the protection of the XFI monitoring code is achieved by combining SFI and CFI. Finally, WIT [16] (Write Integrity Testing) provides protection from memory corruption attacks by verifying whether targets of write operations are valid by comparing with a statically and dynamically defined color table. Since write operations also encompass control-data, it provides integrity of monitoring code as well as protection from control-flow attacks without requiring CFI.

Most of these approaches insert the monitoring code in the application at compile-time. Moreover, the monitoring code can have the same privilege as the attack code and still work because the idea of these approaches is not to allow the malicious code to execute in the first place. Although in-lined methods are extremely fast,they are not adequate in a threat model where the entire OS can be compromised and the monitor code needs to be at a higher privilege than the malicious code.

2.3.2 Passive External Monitoring Approaches

Virtualization technology has played an important role in systems security. The security benefits gained by using virtualization has been first studied in [78, 92]. Several approaches have then been proposed to use virtual machines for providing security in production systems [61, 62, 82, 115]. In general, these approaches utilize the advantage of isolation from the guest VM, and monitor some properties in the guest system to provide security. The passive approaches that use hypervisors and introspection detect attacks after they have occurred since the monitoring is not

initiated by a particular event happening inside the monitored host.

System Virginity Verifier [124] is a rootkit detection approach that validates the kernel code and examine the kernel data (including hooks) known to be the targets of current rootkits. Copilot [114] uses a separate hardware-based solution where a trusted addin PCI card observes the runtime OS memory image and infers possible rootkit presence by detecting any kernel code integrity violations. This work is extended to examine other types of violations such as kernel data semantic integrity in [113]. SecVisor [126] is a tiny hypervisor that uses hardware support to enforce kernel code integrity. The goal is to only allow approved code to be ever executed at the kernel level in the system. This ensures that code injection attacks in the kernel do not work. State-based control-flow integrity (SBCFI) [115] passively checks control-flows in the guest VM at regular intervals to identify whether persistent rootkits were installed in the system. Since monitoring the entire control-flow of the kernel requires noticeable execution time, SBCFI's monitor is invoked at regular intervals to be to reduce overhead on a production system. This approach can detect persistent control-flow changes well after the change occurs.

2.3.3 Active External Monitoring

While passive monitoring has been widely used in the past, Lares [110] recently proposed the method of actively monitoring events in a guest VM. Lares provides the framework of inserting hooks inside the guest OS that can invoke a security application residing in another VM when a particular event occurs. The design of Lares enables complex and large security tools such as Antivirus programs [143] or intrusion detection systems to run on the security VM. The benefit is getting the isolation from the operating system environment that it is protecting. In other words, no malware residing in that environment can access or corrupt the security tool. Second, the invocation of the tool can be controlled and the probes can also be protected by the

hypervisor. All this can be achieved with the flexibility of actively monitoring the system.

However, the cost in communicating an event notification from one VM to another via the hypervisor makes it inappropriate for use in fine-grained active monitoring. Traditional antivirus programs and security tools need to monitor and intercept a wide range of events on the host. Such requirements can severely affect the performance and the incurred overhead may make it practically unusable. Our approach discussed in Chapter 7 overcomes these limitations.

2.3.4 Hypervisor-based Active Monitoring Inside Host

Some approaches use hypervisors and provide active monitoring while the protecting code resides in the host. However, they only provide a limited form of functionality that is not extendable or dynamic in nature that is required by complete antivirus programs. Patagonix [91] provides hypervisor support to detect covertly executing binaries. The system can ensure that only memory pages containing code that is allowed can execute and at the same time unwanted code is neutralized. Similarly, NICKLE [119] mandates that only verified kernel code will be fetched for execution in the kernel space. However, all these systems do not protect kernel hooks from being modified to compromise kernel control flow.

Hooksafe [158] provides a method of fine-grained control-flow checks inside the kernel. Unlike SBCFI, the checks are active in nature, allowing attacks to be detected before they occur. However, like SBCFI, Hooksafe targets control-flow inside the kernel that are persistent in nature. The approach achieves low overhead of around 6% by instrumenting the kernel code to check with addresses that are protected by the hypervisor by relocating the addresses to a separate page that can be protected. Although Hooksafe is a good example of an efficient approach that uses hypervisor to actively detect attacks, it is only limited to control-flow checks. An antivirus

program can have arbitrary data belonging to it that may require protection from unwanted modification. Our approach called SIM, presented in chapter 7 achieves this. However, the techniques of Hooksafe can be used to protect hooks and execution leading to hooks that can activate an external or a secure internal monitor code, providing additional security to both secure active out-of-VM and in-VM approaches.

CHAPTER III

ENABLING STATIC MALWARE ANALYSIS

3.1 Motivation

Dynamic analysis based approaches have arguably offered a better track record and mind share among those researchers working on malware binary analysis. Part of that success is attributable to the challenges of overcoming the formidable obfuscation techniques [139, 143], or packers [132] that are widely utilized by contemporary malware authors. These obfuscation techniques, including function and API call obfuscation, and control flow obfuscations along with a gamut of other protections proposed by the research community [34, 102], have been shown to deter static analyses. While defeating these obfuscations is a prerequisite step to conducting meaningful static analyses, they can largely be overcome by those conducting dynamic analyses. Although dynamic analysis provides only a partial "effects oriented" profile of the full potential of a given malware binary, multipath exploring dynamic analysis [100] has the potential to improve traditional dynamic analysis by executing code paths for unsatisfied trigger conditions. However, the approach cannot guarantee completeness. If successful, static analysis can provide comparable completeness at the fraction of the cost.

Static program analysis can provide complementary insights to dynamic analyses in those occasions where binary obfuscations can be sufficiently overcome. Static program analysis offers the potential for a more comprehensive assessment and correlation of code and data of the program. For example, by analyzing the sequence of invoked system calls and APIs, performing control flow analysis, and tracking data segment references, it is possible to infer logical code bombs, temporal triggers, and

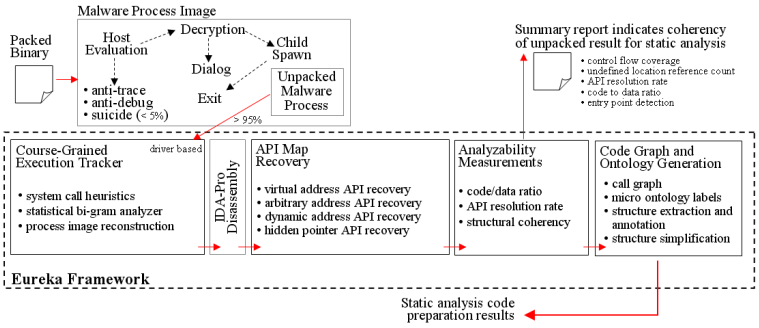


Figure 2: The Eureka Malware Binary Deobfuscation Framework

other malicious system interactions, and from these form higher level semantics about malicious behavior. Features such as the presence of network communication logic, registry and OS manipulations, object creations (*e.g.*, files, processes, inter-process communication) can be detected, whether these capabilities are exercised at runtime or not. Static analysis, when presented with a deobfuscated binary can complement and even *inform* dynamic program analyses with a more comprehensive picture of the program logic.

While a large number of obfuscations continue to reduce the effectiveness of directly attempting to use static analysis, techniques such as unpacking can successfully remove some of the obfuscations. There has been substantial work in automated unpacking techniques [76, 122, 96] before. However, these techniques require fine-grained instruction level execution, which gives us an opportunity to explore more efficient techniques that significantly reduces the time taken to unpack a malware and put it to further analysis. In this chapter, we introduce a malware binary deobfuscation framework referred to as *Eureka*, designed to efficiently facilitate static code analysis. Eureka's coarse-grained execution based unpacking method is suitable for *single-pass* and *multi-pass* unpacking methods, and Eureka also helps identify system call (or Win32 API call) points in the malware by performing API resolution.

3.2 The Eureka Framework

Figure 3.2 presents an overview of the modules and logical work flow that compose the Eureka framework. The Eureka workflow begins with the subject-packed malware binary, which is executed in a VM managed by Eureka. After interrogating local environment for evidence of tracing or debugging, the malware process enters a phase of unpacking and the eventual spawning of its core malware payload logic while a parallel Eureka kernel driver tracks the execution of the malware binary, periodically evaluating the process for signs that it has unpacked its image. In Section 3.4, we present Eureka's course-grained execution tracking algorithm and introduce novel binary n-gram statistical trigger for evaluating when the unpacked process image has reached a stable state. Once the execution tracker triggers a process image dump, Eureka employs the IDA-Pro disassembler [5] to disassemble the image, and then proceeds to conduct API resolution and prepare the code image for static analysis. In Section 3.5, we discuss Eureka's API map recovery module, which provides several automated deobfuscation procedures to recover hidden API invocations that are commonly used to thwart static analysis. Once API resolution is completed, the code image is processed by Eureka's analyzability metrics generation module which compares several attributes to decide if static analysis of the unpacked image yields useful results. Following the presentation of the Eureka framework, we further present a corpus evaluation (Section 3.7) to illustrate the usage and effectiveness of Eureka.

3.3 Previous Work

The problem of obfuscated malware has confounded analysts for decades [143]. The first obfuscation techniques exhibited by malware in the wild include viral metamorphism [143] and polymorphism [139]. Several obfuscation approaches have since been presented in the literature including, opaque predicates [34] and recently opaque constants [102]. Packers and executable protectors [132] are often used to automatically

System	Monitoring Environment	Monitoring Granularity	Trigger Types	Child Process Monitoring	Output Layers	Execution Speed	Potential Evasions
PolyUnpack	Inside VM	Instruction	Model-based	No	1	Slow	1,2,3
Renovo	Emulator	Instruction	Heuristic	Yes	many	Slow	2,4
OmniUnpack	Inside VM	Page	Heuristic	No	many	Fast	2,3
Eureka	Inside VM	System Call	Statistical	Yes	1,many	Fast	2,3

Table 1: Design space of unpackers. Evasions: (1) multiple packing, (2) partial code revealing multi-layered packing, (3) vm detection, (4) emulator detection

add several layers of protection to malware executables. Recent packers and protectors also incorporate API obfuscations that make it hard for analyzers to identify system calls or calls to Windows APIs.

Automated unpacking: There have been several recent attempts at building automated and generic tools for unpacking malware, most notably PolyUnpack [122], Renovo [76], and OmniUnpack [96]. Table 1 summarizes the design space of automated unpackers that illustrates their strengths, differences, and common weakness. PolyUnpack, which was the first automated unpacking technique, builds a static model of the program and uses fine-grained execution tracking to detect when an instruction an instruction outside of the model is executed. PolyUnpack uses the Windows debugging API to single-step through the process execution. Like PolyUnpack, Renovo uses a fine-grained execution monitoring approach to track unpacking progress and considers the execution of newly written code as an indicator of unpack completion. Renovo is implemented using the QEMU emulator, which resides outside the execution environment of the malware and supports multiple layers of unpacking. OmniUnpack is most similar to Eureka in that it uses a coarse-grained execution tracking approach. However, their granularities are orthogonal: OmniUnpack tracks execution at the page level while Eureka tracks execution at the system call level. OmniUnpack uses page-level protection mechanisms available in hardware to identify when code is executed from a page that was newly modified.

Static and dynamic malware analysis: Previous work in malware analysis that uses static analysis has primarily focused on malware detection approaches. Known malicious patterns are identified in [40]. The approach of using semantic behavior to thwart some specific code obfuscations was presented in [41]. Rootkit

behavior detection was presented in [84], and [79] uses a static analysis approach to identify spyware behavior in Browser Helper Objects. Traditional program analysis techniques [104] have been investigated for binary programs in general and malware in particular. Dataflow techniques such as Value Set Analysis [21] aim at recovering the set of possible values that each data object can hold at each program point. CWSandbox [36] and TTAnalyze [24] are dynamic analysis systems that execute programs in a restricted environment and observe sequence of system interactions (using system calls). Pararoma [168] uses system-wide taint propagation to analyze information flow, which it uses for detecting malware. Bitscope [31] incorporates symbolic execution-based static analysis to analyze malicious behavior.

Statistical analysis: Fileprint analysis [140] studies statistical binary content analysis as a means to identify malicious content embedded in files, finding that n-gram analysis is a useful means to detect anomalous file segments. A further finding is that normal system files and malware can be well classified using 1-gram and 2-gram analysis. While our methodology is similar, the problem differs in that we use bi-grams to model unpacked code and it is independent of the code being malicious. N-gram analysis has also been used in other contexts, including anomalous packet detection in network intrusion detection systems such as PAYL [155] and Anagram [154].

3.4 Coarse-grained Execution-based Unpacking

In general, all of the current methods for binary unpacking start with some sort of dynamic analysis. Unpacking systems begin their processing by executing the malware binary, allowing it to self-decrypt its malicious payload logic and to then fork control to this newly revealed program logic. One primary method by which unpacking systems distinguish themselves is in the approach each takes to monitor the progression of the packed binaries' self-decryption process. When the unpacker determines that the process has sufficiently revealed the malicious payload logic, it

will then dump the malicious process image for use in static analysis.

Much of the variability in unpacking strategies comes from the granularity of monitoring that is used to track the self-decryption progress of the packed binary. Some techniques rely on tracking the progress of the packed process on a per-individual instruction basis. We refer to this instruction-level monitoring as *fine-grained* monitoring. Other strategies use more coarse-grained monitoring, such as OmniUnpack, which checkpoints the self-decryption progress of the malicious binary via intercepting interrupts from the page-level protection mechanisms. Eureka, like OmniUnpack, tracks the execution progress of the packed binary image via coarse-grained check pointing. However, rather than using page interrupts, Eureka tracks the malicious process via the system call interface. Eureka's coarse-grained execution tracker operates as a kernel driver that dumps the malicious process image for disassembly when it believes that the malicious payload logic has been sufficiently revealed. In the following, we present two different methods for deciding when to dump the malicious process image, *i.e.,* a heuristic-based method which works for most contemporary malware and a statistical n-gram anlaysis method which is more robust.

3.4.1 Heuristics-based unpacking

Eureka's principal method of unpacking is to follow the execution of the malware program by tracking its progress at the system call level. Among the advantages of this approach, the progression of the self-decrypting process image can be tracked with very little overhead. Each system call indicates that a particular interesting event is occurring in the executing malware. Eureka employs a Windows-driver-based unpacker that hooks the Windows SSDT (System Service Dispatch Table). The driver executes a callback routine when a system call is invoked from a user-level program. We use a filtering approach based on the process ID (PID) of the process invoking the system call. A user-level program initiates the execution of the malware

and informs the Eureka driver of the malware's PID.

The heuristics-based unpacking approach of Eureka exploits a simple strategy in which it uses the event of program exit as triggering the snapshot of the malware's virtual memory address space. That is, the system call `NtTerminateProcess` is used to trigger the dumping of the malware process image, under the assumption that the use of this API implies that the unpacked malicious payload has been successfully decrypted, spawned, and is now ending. Another noticeable behavior we found in a large number of malware programs was that the malware spawns its own executable as another process. We believe this is a widely used technique that detaches from debuggers or system call tracers that trace only the initial malware process. Thus, Eureka also employs a simple heuristic that dumps the malware during the execution of the `NtCreateProcess` system call, we found that a large fraction of current malware programs were successfully unpacked.

A problem with the above heuristic is that not all malware programs exit and keep an executing version resident in memory. There are several weaknesses in this simple heuristics-based approach. Although the above two heuristics may work for a large fraction of malware today, it may not be the same for future malware. With the knowledge of these heuristics, packers may incorporate the features of including process creation as part of the unpacking process. This would mean that unpacking may not have completed when the `NtCreateProcess` system call is intercepted. Also, malware authors can simply avoid exiting the malware process, avoiding the use of the `NtTerminateProcess` system call. Nevertheless, these very basic and very efficient heuristics demonstrate that very simple and straightforward mechanisms can be effective in unpacking a significant fraction of today's malware (as much as 80% of malware analyzed in our corpus experiments, Section 3.7). Where these heuristics fail, our statistical-based n-gram strategy provides a more than sufficient complement to unpack the remaining malware.

3.4.2 Statistics-based unpacking

As an alternative to its system-call heuristics, Eureka also tracks the statistical distribution of executable memory regions. In developing such an approach, we are motivated by the simple premise that unpacked executables have fundamentally different statistical properties that could be exploited to determine when a malware program has fully unpacked itself. A Windows PE (portable executable) is composed of several different types of regions. These include file headers and data directories, code sections (typically labeled as .text), and data sections (typically labeled as .data). Intuitively, as the malware unpacks itself, we expect that the code-to-data ratio would increase. So we expect that tracking the volume of code and data in the executable would provide us with a measure of the progress of unpacking. However several potential complications could arise that must be considered:

- Code and data are often interleaved, especially in malicious executables.
- Data directory regions such as import tables that have statistically similar properties to data sections (*i.e.,* ASCII data) are embedded within code sections.
- Properties of data sections holding packed code might vary greatly based on packers and differ significantly from data sections in benign executables.

To address these issues, we develop an approach that models statistical properties of unpacked code. Our approach is based on two observations. First, code has certain intrinsic properties that tend to be invariant across executables (*e.g.,* certain opcodes, registers, and instruction sequences are more prevalent than others). These statistical properties may be used to measure relative changes in the volume of unpacked code. Second, we expect that the volume of unpacked code would be strictly increasing as a packed malware executes and unravels itself. Surprisingly, we find that both our assertions hold for the vast majority of malware and across most packers.

Mining statistical patterns in x86 code: As a means to study typical and frequently occurring patterns in x86 code, we began by looking at a small collection of benign PE executables. A natural way to search for such patterns is to use a simple n-gram analysis. Specifically, we were interested in using n-gram analysis to build models of sections of these executables that contained x86 instructions. Our first approach was to simply extract entire sections from the PE header that was labeled as code. However, we found that large portions of these sections also contained long sequences of ASCII data from non x86 instructions, *e.g.*, data directories or DLL names, which biased our analysis. To alleviate this bias, we used the IDA Pro disassembler, to extract regions from these executables that were marked as functions by looking for arguments to the MakeFunction calls in the IDC file. We then performed bigram analysis on this data. We chose bigrams because x86 opcodes tend to be either 1-byte or 2-bytes. By looking at frequently occurring bigrams we are looking at the most common opcode pairs or 2-byte opcodes. Once we developed a list of the most common bigrams for the benign executable, we used `objdump` output to evaluate whether bigrams occur in opcodes or operands (addresses, registers). Intuitively, one expects the former to be more reliable than the latter. We provide a summary in Table 3.4.2. Based on this analysis, we selected FF 15 (pushl) and FF 75 (call) as two candidate bigrams that are prevalent in x86 code. We also looked for spaced bigrams (byte pairs separated by 1 or more bytes). We found that the call instruction with one byte opcode (e8) has a relative offset. The last byte of this offset invariably ends up being 00 or FF depending on whether has a positive or negative offset. Thus high frequencies of e8 _ _ _ 00 and e8 _ _ _ ff are also indicative of x86 code.

To evaluate the feasibility of this approach, we examined bigram distributions on a corpus of 1291 malware instances. We first unpacked each of these instances using our heuristic-based unpacker and then evaluated the quality of unpacking by evaluating

Table 2: Occurrence summary of bigrams

Bigrams	calc (117 KB)	explorer (1010 KB)	ipconfig (59 KB)	lpr (11 KB)	mshearts (131 KB)	notepad (72 KB)	ping (21 KB)	shutdown (23 KB)
FF 15 (call)	246	3045	184	24	192	415	58	132
FF 75 (push)	235	2494	272	33	274	254	41	63
E8 - - - 0xff (call)	1583	2201	181	19	369	180	87	49
E8 - - - 0x00 (call)	746	1091	152	62	641	108	57	66

the code-to-data ratio in an IDA Pro disassembly. We found that the heuristic-based unpacker did not produce a useful unpacking in 201 instances (small amount of code and low code-to-data ratio in the IDA disassembly). Out of the remaining 1090 binaries, we labeled 125 binaries as being originally unpacked (significant amount of code and high code-to-data ratio in both packed and unpacked disassemblies) and 965 as being successfully unpacked (significant amount of code and high code-to-data ratio only in the disassembly of the unpacked executable). Using counts of aforementioned bigrams, we were able to produce output consistent with that of IDA disassembly evaluation. We correctly identified all 201 instances of still-packed binaries, all 125 instances of originally unpacked binaries, and 922 (out of 965) instances of the successfully unpacked binaries. In summary, this simple bigram counting approach had over a 95% success rate in distinguishing between packed and unpacked malware instances.

STOP – Statistical Test for Online unPacking: Inspired by the results from offline bigram counting experiments, Eureka incorporates STOP, an online algorithm for determining the terminating (or dumping) condition. We pose the problem as a simple hypothesis testing argument that checks for increase in mean value of bigram counts. Our null hypothesis is that the mean value of x86 instruction bigrams has not increased. We would like to conclude that the mean value has increased when we see a consistent and significant shift in the bigram counts. Let us assume that we have the prior mean (μ_0) for the candidate x86 instruction bigrams, and that we have a sample of N recent bigram counts. We assume that this sample is normally distributed with mean value (μ_1) and standard deviation (σ_1). We compute $z0 = \frac{\mu_1 - \mu_0}{\sigma_1}$. If $z0 > 1.645$ then we reject the null hypothesis (with a confidence level of 0.95 for a normal distribution). We have integrated the STOP algorithm into our

Eureka execution tracking module. STOP parameters include the ability to choose to compute the mean value of particular bigrams at each system call, every n system calls for a given value of n, or only when certain anomalous system calls are invoked.

3.5 API Resolution Techniques

User-level malware programs require the invocation of system calls to interact with the OS in order to perform malicious actions. Therefore, analyzing and extracting malicious behaviors from these programs require the identification of invoked system calls. Besides the predefined mechanism of system calls that require trapping to kernel, application programs may interact with the operating systems via higher level shared helper modules. For example, in Windows, the Win32 API is a collection of services provided by helper DLLs that reside in user space, while the native APIs are services provided by the kernel. In such a design, the user-level API allows a higher-level understanding of behavior because most of the semantic information is lost at the native level. Therefore, an in-depth binary static analysis requires the identification of all Windows API calls, and call sequences, made within the program.

Obfuscations that impede analysis by hiding API calls have become prevalent in malware. Analyzers such as IDA Pro [5] or OllyDbg [6] support the standard loading and linking method of binaries with DLLs, which modern packers bypass Rather, they employ a variety of nonstandard techniques to link or connect call sites with the intended API function residing in a DLL. We refer to the task of deobfuscating or identifying Windows API function targets from the image of a previously packed malware binary, no matter how they are referenced, as *obfuscated API resolution*. In this section, we first provide a background on how normal API resolution occurs in Windows, and then contrast this with how Eureka handles problems of obfuscated API resolution. These analyses are performed on IDA Pro's disassembly of the unpacked binary, as produced by Eureka's automated unpacker.

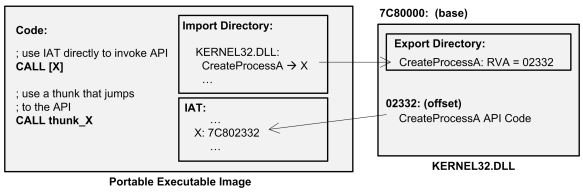


Figure 3: Example of the standard linking mechanism of PE executables in Windows

3.5.1 Background: standard API resolution

Understanding the challenges of obfuscated API resolution first requires an understanding of how packers typically avoid the standard methods of linking API functions that reside in user-level DLLs. The Windows process loader and linker are responsible for linking DLLs with a PE (Portable Executable) binary. Figure 3 illustrates the high-level view of the mechanism. Each executable contains an *import table directory*, which consists of entries corresponding to each DLL it imports. The entries point to tables containing names or ordinals for functions that need to be imported from a specific DLL. When the binary is loaded, the required DLLs are mapped into the memory address space of the application, and the *export table* in the DLL is used to determine the virtual addresses of the functions that need to be linked. A table called the *Import Address Table* (IAT) is filled in by the loader and linker with the virtual addresses of each imported function. This table is referred to by indirect control flow instructions in the program to call the functions in the linked DLL.

3.5.2 Resolving obfuscated APIs without the import tables and IAT

Packers avoid using the standard linking mechanism by removing entries from the import directory of the packed binaries. For the program to function as before after unpacking, the logic of loading the DLLs and linking the program with the API functions is incorporated into the program itself. Among other methods, this may include explicit invocations to `GetProcAddress` and `LoadLibrary` API calls.[1] The `LoadLibrary` API provides a method of mapping a DLL into a process's address

[1]In most cases, at least these two API functions are kept in the import table, or their addresses are hard-coded in the program.

space during execution, and the `GetProcAddress` API returns the virtual address of an API function in a loaded DLL.

Let us assume that the IAT defined in a malware executable's header is incomplete, corrupt, or not used at all. Let us further assume that the unpacking routine may include entries in the IAT that are planted to mislead naive analysis attempts. Moreover, the malware executable has the power to recreate a similar table in any memory location of its choosing or use methods that may not require table-like data structures. The objective of Eureka's API resolution module is to resolve APIs in such cases to facilitate the static analysis of the executable. In the following, we outline the strategies used by the Eureka API resolution module to accomplish these deobfuscations, presented in the increasing order of complexity.

3.5.2.1 Handling DLL Obfuscations

DLLs loaded at standard virtual addresses: By default, DLLs are loaded at the virtual address specified as the image base address in the DLL's PE header. The standard Windows Win32 DLLs specified bases do not clash with each other. Therefore, unless intervened, the loader and linker can load all these DLLs at the specified base virtual addresses. By assuming this is the case, a table of probable virtual addresses of each exported API function from these DLLs can be built. This simple method has been found to work for many unpacked binary malware images. For example, for Windows XP service pack 2, the `KERNEL32.DLL` has a default image base address of `0x7C800000`. The RVA (relative virtual address) of the API `GetProcessId` is `0x60C75`, making its default virtual address `0x7C860C75`.

In such cases, Eureka's analysis proceeds as follows to reconstruct API associations. For each Win32 DLL D_i, let B_i be the default base address. Also, let there be k_i exported API functions, where each function F_i, j has the RVA (relative virtual address) $R_{i,j}$. Eureka builds a database of virtual addresses $V_{i,j} = B_i + R_{i,j}$ and their

corresponding API functions. Whenever Eureka finds a call site c with resolved target address $A(c)$, it searches all V_{i_j} to identify the API function target. We find that this method works as long as the DLLs are loaded in the default base address.

DLLs loaded at arbitrary virtual addresses: To make identification of an API harder, there may be cases where a DLL is loaded into a nonstandard base address by system calls to explicitly map them into a different address space. As a result, the address found during analysis of the unpacked binary may not be found in the computed virtual address set. In this case, we can utilize some of the dynamic information captured by running malware (in many cases, this information can be harvested during Eureka's unpacking phase). The idea is to use runtime information of native system calls that are used to map DLL and modules into the virtual address space of an application. Since our unpacker traces native system calls, we can look for specific calls to `NtOpenSection` and `NtMapViewOfSection`. The former system call identifies the DLL name and the latter provides the base address where it is loaded. Eureka correlates these two calls using the handle returned by the first system call.

3.5.2.2 API resolution for statically identifiable targets

One way to identify an invocation of an API function without relying on the import directory of the unpacked image is by testing targets of call sites to see whether they point to specific API functions. We assume that a call site may use an indirect call or a jump instruction. Such instructions may involve a pointer directly or may use a register that is loaded with an address in an earlier instruction. To identify targets in a generic manner, Eureka uses static analysis on the unpacked disassembly.

Eureka starts by performing control flow analysis on the program. The use of IDA Pro disassembly simplifies analysis by marking subroutine boundaries and interprocedural control flows. Furthermore, control flow instructions that have statically identified targets that reside within the program are also resolved. In addition, IDA

Pro identifies any *valid* API calls through the import directory and the IAT. Eureka's analysis task then is to resolve unknown static or statically resolvable target addresses in control flow instructions. These are potential calls to API functions residing in DLLs. Our algorithm proceeds as follows. First, Eureka identifies functions in the disassembly (marked as subroutines using the `SUB` markers). For each function, the control flow graph is built by identifying basic-blocks as nodes and static intra-procedural control flow instructions that connect them as edges. Eureka then models inter-procedural control flow by observing `CALL` or `JMP` instructions to subroutines that IDA already identifies. It selects any remaining such instructions with an unrecognized target as potential API call sites. For these instructions, Eureka uses static analysis to identify the absolute memory address to which they will transfer control.

We now use a simple notation to express the x86 instructions that Eureka analyzes. Let the set of all instructions be I. For any instruction $i \in I$, we use the notation $S(i)$ as the source operand if one exists, and $T(i)$ as the target operand. The operands may be immediate values, memory pointer indirection or a register. Suppose the set of potential API call instructions is $C \subseteq I$. Our goal is to find the target address of a potential API call instruction c, which we express by $A(c)$. For instructions with immediate addresses, $A(c)$ can be found directly from the instruction. For indirect control transfers using a pointer, such as `CALL [X]`, Eureka considers the static value stored at address X as a target. Since Eureka uses the disassembly generated by IDA, the static value at address X is included as data definition with the name `dword_X`.

For register-based control transfers, Eureka needs to identify the value loaded in the register at the point of initiating the transfer. Some previous instruction can load the register with a value read from memory. A generic way to identify the target is to extract a sequence of instructions that initially loads a value from a specific memory address to a register and subsequently is loaded to the register that is used

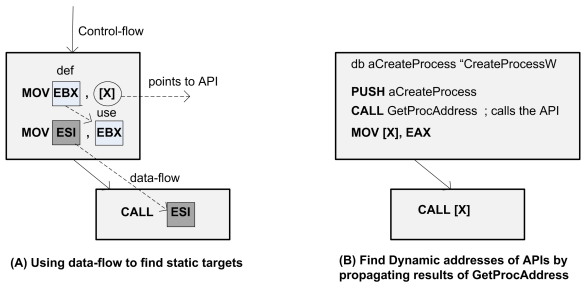


Figure 4: Illustration of Static Analysis Approaches used to Identify API Targets

in the control-transfer instruction. Eureka resorts to dataflow analysis for solving these cases. Using standard dataflow analysis at the intra-procedural level, Eureka identifies *def-use* instruction pairs. A def-use pair (d, u) is a pair of instructions where the latter instruction u uses an operand that is defined in d, and there is a control flow path between these instructions with no other definitions of that operand in between. For example, a `MOV ESI, EAX` followed by `CALL ESI` instruction with no other redefinitions of `ESI` forms a def-use pair for the register `ESI`. To find the value that is loaded in the register at the call site, starting from a potential call site instruction, Eureka identifies a chain of def-use pairs that end at this instruction involving only operands that are registers. Therefore, the first pair in the chain contains a def that loads to a register a value from memory or an immediate value, which is subsequently propagated to the call site. Figure 4(a) illustrates these cases. The next phase is to determine whether the address $A(c)$ for a call site c is indeed an API function, and if so Eureka resolves its API name.

3.5.2.3 API resolution for dynamically computed addresses

In some cases, the resolved target address $A(c)$ can be uninitialized. This may happen if the snapshot is taken at a point during the execution when the resolution of the API address has not taken place in the malware code. It may also be the case that the address is supposed to be returned from a system call such as `GetProcAddress`, and thus is not contained in the unpacked memory image. In such cases, Eureka attempts to analyze the malware code and extract the portion of code that is supposed to

update this address by identifying instructions that write to the memory location that contained $A(c)$. For each of these instructions, Eureka constructs def-use chains and identifies where they are initiated. If in the control flow path there is a call to the `GetProcAddress`, Eureka identifies the arguments pushed onto the stack before calling the service. Since it is one of the arguments, Eureka can directly identify the name of the API whose address is returned and stored in the pointer. Figure 4(b) illustrates a sample code template and how our analysis propagates results of `GetProcAddress` to call sites.

3.6 Evaluation Metrics

We consider the problems of measuring and improving analyzability after API resolution. Although a manual inspection can determine the quality of the output and its suitability for applying static analysis, in a large corpus of thousands of malware programs, automated methods for performing this step are essential. Technically, without the knowledge of the original malware code, it is impossible to precisely conclude how successfully the obfuscations applied to a code have been removed. Nevertheless, several heuristics can aid malware analysts and other post-unpacking static analysis tools in deciding which unpacked binaries can be analyzed successfully, and which require further attempts at deobfuscation. Poor analyzability metrics could further help detect when previously successful malware deobfuscation strategies are no longer successful, possibly due to new countermeasures employed by malware developers to thwart the unpacking logic. Here we present heuristics that we have incorporated in Eureka to express the quality of the disassembled process image, and its potential analyzability in subsequent static analyses.

Code-to-data ratio: An observable difference between packed code and unpacked code is the amount of identifiable code and data found in the binary. Although differentiating between code and data on x86 variable length instructions is a known

hard problem, in practice the state-of-the-art disassemblers and analyzers such as IDA Pro are quite capable of identifying code by recursively passing through code and by taking into account specific valid code sequences. However, these methods tend to err on the side of detecting data as code, rather than the other way around. Therefore, if code is identified via IDA Pro, it can be taken with confidence that it is actual code. The amount of code that is identified in and provided from an unpacker can be used as a reasonable indication of how completely the binary was unpacked. Since there is no ground truth on the amount of code in the original malware binary prior to its packing, we have no absolute measures from which we can compare the quality of the unpacked results. However, empirically, we find that the ratio of code to data found in the unpacked binary is a useful analyzability metric. Usually, any sequence of bytes that is not identified as code is treated as data by IDA Pro. In the disassembled code, these data are represented using the data definition assembler mnemonics — `db, dw` or `dd`. We use the ratio of identified code and data by IDA Pro as an indication of unpacking quality. The challenge with this measurement is in identifying the threshold above which we can conclude that packing was successful. We used an empirical approach to determine a suitable threshold for this purpose. When experimenting with packed and unpacked binaries of benign programs, we observed that the amount of identified code is very low for almost all different packer-generated packed binaries. There were slight variations depending on the unpacking code inserted by the packer. Still, we found the ratio to be well below 3% in all cases. Although the ratio of code vs. data increased significantly after unpacking, it was not equal to the original benign program prior to packing, because the unpacked code still contained the packed data in the memory image, which appeared as data definitions in the disassembly. We found that most of the successfully unpacked disassemblies had code-to-data ratios well above 50%. Eureka uses the 50% threshold as the value of valid unpacking.

API resolution success: When attempting to conduct a meaningful static analysis on an unpacked binary, one of the most important requirements is the proper identification of control flow, whether it relates to Windows APIs or to the malware's internal functions. Incomplete control flow can adversely affect all aspects of static analyses. One of the main culprits of control flow analysis is the existence of indirect control flow instructions whose targets are not statically identifiable and can be derived only by dynamic means. In Section 3.5, our presented API resolution method tries to identify the targets of call sites that were not identified by IDA Pro. If the target is not resolvable, it may be a call to an API function that was successfully obfuscated beyond the reversal techniques used by Eureka, or it may be a dynamically computed call to an internal function. In both cases, we lose information about the control flow behavior from that point in the program. By taking success and failure scenarios into account, we can compute the ratio of resolved APIs and treat it as an indication of quality of subsequent static analysis. Our API resolution quality is expressed as a percentage of total number of API calls that have been resolved from the set of all potential API call sites, which are indirect or register-based calls with unresolved target. A higher value of p indicates that the resulting deobfuscated Eureka binary will be suitable for supporting static analyses that support more in-depth behavioral characterization.

3.7 Experimental Results

We now evaluate the effectiveness of Eureka using three different datasets. First, we measure how Eureka and other unpackers handle various common packers using a dataset of packed benign executables. Next, we evaluate how Eureka performs on two recent malware collections: a corpus of 479 malicious executables obtained from spam traps and a corpus of 435 malicious executables obtained from our honeynet.

Table 3: Evaluation of Eureka, PolyUnpack and Renovo: $\surd$ = unpacked; $\otimes$ = partially unpacked; $\times$ = unpack failed.

Packer	PolyUnpack Unpacking	Renovo Unpacking	Eureka Unpacking	Eureka API Resolution
Armadillo	$\times$	$\otimes$	$\surd$	64%
Aspack 2.12	$\otimes$	$\surd$	$\surd$	99%
Asprotect 1.35	$\otimes$	$\surd$	$\times$	–
ExeCryptor	$\surd$	$\otimes$	$\surd$	2%
ExeStealth 2	$\times$	$\surd$	$\surd$	97%
FSG 2.0	$\surd$	$\surd$	$\surd$	0%
MEW 1.1	$\surd$	$\surd$	$\surd$	97%
MoleBoxPro	$\times$	$\surd$	$\surd$	98%
Morphine 1.2	$\surd$	$\otimes$	$\surd$	0%
Obsidium	$\times$	$\times$	$\surd$	99%
PeCompact 2	$\times$	$\surd$	$\surd$	99%
Themida	$\times$	$\otimes$	$\otimes$	–
UPX 3.02	$\surd$	$\surd$	$\surd$	99%
WinUPack 3.99	$\otimes$	$\surd$	$\surd$	99%
Yoda 3.53	$\otimes$	$\otimes$	$\surd$	97%

3.7.1 Benign dataset evaluation: Goat test

We evaluate Eureka using a dataset of packed benign executables. Specifically, we used several common packers to pack an instance of the popular Microsoft Windows executable, `notepad.exe`. An advantage of testing with a dataset of custom-packed benign executables is that we have ground truth for what the malware is packed with and we know exactly what is expected after unpacking. This makes it easier to evaluate the quality of unpacking results. We compare the unpacking capability of Eureka to that of PolyUnpack (using a limited distribution version obtained from the author) and Renovo (by submitting to BitBlaze malware analysis service [28]). We were unable to acquire OmniUnpack for our test results.

These results are summarized in Table 3. In cases where an output was found, we used Eureka's code-to-data ratio heuristic to determine whether it was successfully unpacked and manually also verified the results of the heuristic. For Renovo, we compare with the last layer that was produced in the case of multiple unpacked layers. The results show that Eureka performs well compared to other unpacking solutions. Eureka was successful in all cases except Asprotect, which interfered with Eureka's driver, and Themida, where the output was an altered unpacking with API calls emulated. In Figure 5, we illustrate how the bigram counts change as Eureka

executes for three of the packers. We find that in most cases the bigram counts change synchronously and very sharply (similar to ASPack) making it easy to determine appropriate points for snapshotting execution images. We find that Eureka is also robust to packers that naively employ multiple layers such as MoleBox and some incremental packers such as Armadillo.

In this comparison study, PolyUnpack failed in many instances including cases where it just unveiled a single layer of packing while the output still remained packed. We suspect that aggressive implementation of anti-debugging features might be impairing its current success. Renovo, on the other hand, provided several unpacked layers in all cases except for Obsidium. Further analysis of the output however revealed that in some cases the binary was not completely unpacked. Finally, our results show that Eureka's API resolution technique was able to determine almost all APIs for most packers and failed considerably in some others. Particularly, we found ExeCryptor and FSG to use a large amount of code rewriting for obfuscating API calls, including use of arbitrary combinations of complex instruction sequences to dynamically compute the targets.

3.7.2 Malicious data set evaluation

Spam corpus evaluation: We begin by evaluating how Eureka performs on a corpus of 481 malicious executables obtained from spam traps. The results are very encouraging. Eureka was able to successfully unpack 470 of 481 executables. Of the 470 executables from this spam corpus, 401 were successfully unpacked simply using the heuristic-based unpacker, the remainder could only be unpacked using Eureka's bigram statistical hypothesis test. We summarize Eureka's results in Tables 4 and 5. Table 4 illustrates the various packers used (as classified by PeID) and describes how effectiveness of Eureka varies across packers. Table 5 classifies the dataset based on antivirus (AV) labels obtained from Virus-Total [149] illustrating how Eureka's

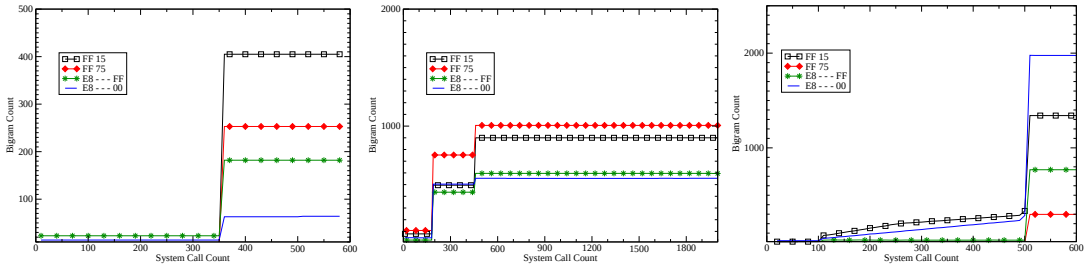


Figure 5: Bigram counts during execution of goat file packed with Aspack(left), Molebox(center), Armadillo(right).

Table 4: Eureka performance by packer distribution on the spam malware corpus.

Packer	Count	Eureka Unpacking	Eureka API Resolution
Unknown	186	184	85%
UPX	134	132	78%
Warning:Virus	79	79	79%
PEX	18	18	58%
MEW	12	11	70%
Rest (10)	52	46	83%

Table 5: Eureka performance by malware family distribution on the spam malware corpus.

Malware Family	Count	Eureka Unpacking	Eureka API Resolution
TRSmall	98	98	93%
TRDldr	63	61	48%
Bagle	67	67	84%
Mydoom	45	44	99%
Klez	77	77	78%
Rest(39)	131	123	78%

effectiveness varies across malware families and validating the quality of Eureka's unpacking.

Honeynet corpus evaluation: Next, we evaluate how our system performs on a corpus of 435 malicious executables obtained from our honeynet deployment. We found that 178 were packed with Themida. In these cases, Eureka is only able to obtain an altered execution image.[2] These results highlight the importance of building better analysis tools that can deal with this important problem. Out of the remaining 257 binaries, 20 were binaries that did not execute on Windows XP (either because

[2]As we see from Table 3, this class of packers also poses a problem for the other unpackers

Table 6: Eureka performance by packer distribution on the honeynet malware corpus minus Themida.

Packer	Count	Eureka Unpacking	Eureka API Resolution
PolyEne	109	109	97%
FSG	36	35	94%
Unknown	33	29	67%
ASPack	23	22	93%
tElock	9	9	91%
Rest(9)	27	24	62%

Table 7: Eureka performance by malware family on the honeynet malware corpus minus Themida.

Malware Family	Count	Eureka Unpacking	Eureka API Resolution
Korgo	70	70	86%
Virut	24	24	90%
Padobot	21	21	82%
Sality	17	17	96%
Parite	15	15	96%
Rest(19)	90	81	90%

they were corrupted or because we could not determine the right execution environments). Eureka is able to successfully unpack 228 of the 237 remaining binaries and produce successful API resolutions in most cases. We summarize results of analyzing the remaining 237 binaries in Tables 6 and 7. Table 6 illustrates the distribution of the various packers used in this dataset (as classified by PeID) and describes how effectiveness of Eureka varies across the packers. Table 7 classifies the dataset based on AV labels obtained from Virus-Total and illustrates how the effectiveness of Eureka varies across malware families.

3.8 Summary

We have presented the Eureka malware deobfuscation framework, to assist in the automated preparation of malware binaries for static analysis. Eureka distinguishes itself from existing unpacking systems in several important ways. First, it introduces a new methodology for automated malware unpacking, using coarse-grained NTDLL system call monitoring. The system provides support for both statistical and heuristic-based unpacking triggers and allows child process monitoring. Second

Eureka includes an API resolution system that is capable of overcoming several contemporary malware API address obfuscation strategies. Finally, Eureka includes an analyzability assessment module, simplifies graph structure and automatically generates and annotates nodes in the call graph with ontology labels based on API calls and data references. While the post-unpacking analyses are novel to our system, they are complementary and could be integrated into other unpacking tools.

Our results demonstrate that Eureka successfully unpacks majority of packers (13 of 15) and that its performance is comparable to other automated unpackers. Furthermore, Eureka is able to resolve most API references and produce binaries that result in analyzable disassemblies. We evaluate Eureka on two collections of malware: a spam malware corpus and a honeynet malware corpus. We find Eureka is highly successful in unpacking the spam corpus (470 of 481 executables), reasonably successful in unpacking the honeynet corpus (complete dumps for 228 of 435 executables and altered dumps for 178 of 435 executables) and produces useful API resolutions. Finally, our runtime performance results validate that the Eureka workflow is highly streamlined and efficient, capable of unpacking more than 90 binaries per hour. Eureka is now available as a free Internet service at `http://eureka.cyber-ta.org`.

Although the unpacking system is flexible and fast, it uses a kernel module to aid in tracking system calls. In order to improve the *robustness* of the presented implementation, the techniques in the next chapter to make dynamic analysis robust can be utilized to help thwart detection attacks that may detect Eureka's unpacker. Overall, Eureka shows how malware obfuscations of specific classes may be removed efficiently to enable static analysis. There are large number of obfuscations that it cannot remove, including anti-disassembly tricks, opaque predicates, page-by-page packing, etc. Since malware authors continue to use more sophisticated obfuscations, the effectiveness of Eureka can keep on reducing over time. However, since it is still an efficient method, it is possible to use this as a first layer of analysis that can quickly

select malware samples from the entire pool that can be deobfuscated quickly. The rest of the malware can then be sent to other more comprehensive analysis schemes.

CHAPTER IV

ROBUST DYNAMIC MALWARE ANALYSIS

4.1 Motivation

Recent advances in malware analysis [85, 42, 20, 43] show promise in understanding modern malware, but before these and other approaches can be used to determine what a malware instance does or might do, the runtime behavior of that instance and/or an unobstructed view of its code must be obtained. However, malware authors are incentivized to complicate attempts at understanding the internal workings of their creations. Therefore, modern malware contain a myriad of anti-debugging, anti-instrumentation, and anti-VM techniques to stymie attempts at runtime observation [143, 18]. Similarly, techniques that use a malware instance's static code model are challenged by runtime-generated code, which often requires execution to discover.

In the obfuscation/deobfuscation game played between attackers and defenders, numerous anti-evasion techniques have been applied in the creation of robust in-guest API call tracers and automated deobfuscation tools [147, 123, 163, 97]. More recent frameworks [1, 146] and their discrete components [24] attempt to offer or mimic a level of transparency analogous to that of a non-instrumented OS running on physical hardware. However, given that nearly all of these approaches reside in or emulate part of the guest OS or its underlying hardware, little effort is required by a knowledgeable adversary to detect their existence and evade [56, 118].

In this chapter, we present a *transparent*, *external* approach to malware analysis. Our approach is motivated by the intuition that for a malware analyzer to be transparent, it must not induce any side-effects that are *unconditionally detectable* by its observation target. In formalizing this intuition, we model the structural properties

and execution semantics of modern programs to derive the requirements for transparent malware analysis. An analyzer that satisfies these transparency requirements can obtain an *execution trace* of a program identical to that if it were run in an environment with no analyzer present. Approaches unable to fulfill these requirements are vulnerable to one or more *detection attacks*–categorical, formal abstractions of detection techniques employed by modern malware.

Creating a *transparent* malware analyzer required us to diverge from existing approaches that employ in-guest components, API virtualization or partial or full system emulation, because none of these implementations satisfy all the transparency requirements. Based on novel application of hardware virtualization extensions such as Intel VT [145], our analyzer–called *Ether*–resides completely outside of the target OS environment– there are no in-guest software components vulnerable to detection or attack. Additionally, in contrast to other external approaches, the hardware-assisted nature of our approach implicitly avoids many shortcomings that arise from incomplete or inaccurate system emulation.

4.2 Previous Work

Traditionally, anti-virus scanners have used simple emulation and API virtualization in their scanning engines [143]. More recent malware analysis efforts make heavy use of virtualized and emulated environments for their operation. Examples include systems derived from the BitBlaze project (e.g., Polyglot [37] and Panorama [168]), Siren [29] and others. Honeypot-based projects also employ virtual environments for trapping and investigating malware [73, 156, 117].

Virtualization- or emulation-based approaches can be used to construct malware processing systems that provide a level of isolation between the guest and the host operating systems. In these systems, modifications made to the guest by an instance of malware can be quickly discarded, ensuring that each instance runs in the same

sterile environment. Approaches that employ these ideas to obtain scalability include malware analysis services such as Norman Sandbox [11], CWSandbox [163] and Anubis [1].

Previous frameworks for fine-grained tracing of programs include VAMPiRE [147], BitBlaze and Cobra [146]. Among these, VAMPiRE is in-guest, BitBlaze uses whole-system emulation, while Cobra traces malware at the same privilege level as itself. None of these of these frameworks use hardware virtualization extensions for their analysis capabilities or information gathering. Automated unpackers–common applications for fine-grained analysis–include PolyUnpack [123], Renovo [77], and OmniUnpack [97]. Of these, only Renovo takes an out-of-guest approach, utilizing whole-system emulation as a core part of its unpacking engine. PolyUnpack and OmniUnpack use in-guest approaches for their unpacking and hence run at the same privilege level as the malware they are analyzing.

Frameworks which could be used for system call or Windows API tracing include Detours [69] and DynInst [3]. System call tracing using out-of-guest environments has been previously implemented in VMScope [72] and TTAnalyze [24], both of which are based off QEMU [26]. There are many in-guest approaches used to trace Windows API functions, which include older tools such as FileMon [4], RegMon [7], and more recently sandboxing environments such as CWSandbox and Norman Sandbox. These approaches use a combination of API and/or API virtualization, which are detectable by malware running at the same privilege level [57].

4.3 A Formal Framework

In this section, we analyze the requirements for building a dynamic *transparent* malware analysis system (i.e., one that cannot be detected and evaded by the malware being analyzed). These requirements serve as the guiding principles to the design and implementation of *Ether* (Section 4.4). We start with a simple and abstract model

for program execution and present the basic definition and theorem of transparent malware analysis. We then extend our formal framework to consider virtual memory, privilege levels, system calls, exception handling, and execution timing that are part of realistic program execution environments.

4.3.1 Abstract Model of Program Execution

We model a program's execution at the machine instruction level. Since a low level instruction can access memory and CPU directly, we consider a system state as the collection of the contents of memory and CPU registers. Let M be the set of all possible memory states, C be the set of all possible CPU states, and I be the set of all possible instructions. We model any program P as a tuple (I_P, D_P) of code and data, where $I_P \subseteq I$ is the set of all instructions belonging to the program, including any dynamically generated instructions, and D_P is the set of static data used by the code. The surrounding runtime environment E during the execution of P contains the operating system, the underlying hardware, external inputs, etc. In order to obtain an execution trace of P in E, we need the subsequence of all the instructions executed in E that belong to I_P. For this reason, we define a transition function $\delta_{E,I_P} : I_P \times M \times C \rightarrow I_P \times M \times C$ that transitions among instructions belonging to I_P. A program is initiated by loading the static code and data into the memory state $m_0 \in M$. Assume that the program starts executing from a specified initial instruction $i_0 \in I_P$ with the initial CPU state $c_0 \in C$. The trace of the program P in E is $T(P, E)$, which is an ordered set defined as $T(P, E) = (i_0, i_1, i_2, ..., i_l)$, where $(i_k, m_k, c_k) \xrightarrow{\delta_{E,I_P}} (i_{k+1}, m_{k+1}, c_{k+1})$ for $0 \leq k < l$.

4.3.2 Transparent Malware Analysis

Suppose P is a malware program. Consider a dynamic malware analyzer P_A whose goal is to learn about P's activities (e.g., its execution traces. P_A or at least some of its components need to reside in the environment E). The malware program P, on

the other hand, will try to detect the presence of P_A and change or hide its intended activities. Thus, we want P_A to be *transparent*, or, undetectable by P.

Since P_A and P share at least some resource in the run-time environment (e.g., the CPU), *covert channel* can exist that leaks information about the presence of P_A to the malware P. To prevent such leakage, the principle of *non-interference* [27] dictates that the execution of P_A *shall not interfere* with the execution of P. Intuitively, if non-interference is achieved, P has the same execution (for the same given input) regardless of P_A.

We model the attempt by P to detect the presence of P_A in E as $d_P(E)$. That is, since the operating system generally isolates one program execution from the other, P must perform its own d_P instructions (included as part of its code I_P, i.e., $d_P \subset I_p$) to get/infer information about P_A. For example, d_p can include an instruction to query the debugging flag in E. Without loss of generality, we assume that P will alter its execution path when it detects P, i.e., if $d_P(E) = 1$, because the malware author would try to hide the behavior of malware from an analyzer. That is, P's execution becomes: $\delta_{E,I_P}(i_k, m_k, c_k) = (i'_{k+1}, m'_{k+1}, c'_{k+1})$ if $d_P(E) = 1$, i.e., P executes a different next instruction i'_{k+1} instead of the intended i_{k+1} when P_A is detected.

Before P_A is enabled in E, we have $d_p(E) = 0$. Denote the environment with P_A running as A. The transparency goal is to achieve $d_P(A) = d_P(E) = 0$, so that even when P_A is analyzing its behavior, P still executes the same instructions. Thus, we have the following definition:

Definition 1. *Assume E is a runtime environment, and A is E with malware analyzer P_A running. P_A is transparent if for any malware P, $d_p(A) = d_p(E) = 0$.*

The above definition provides a starting point for analyzing the requirements for a transparent malware analyzer. Ultimately, the goal of a *transparent* malware analyzer is to extract the same execution traces from malware as if the analyzer is not present. Thus, we have the following malware analysis theorem:

Theorem 1. *If E is a runtime environment and A is the same environment with the addition of a malware analyzer P_A, then P_A is transparent if and only if $T(P, E) = T(P, A)$ for any malware program $P(I_P, D_P)$.*

Proof. In both environments E and A the same external inputs are provided to the program P since they are modeled as part of the environments. Let $m_{E,0}$ and $c_{E,0}$ be the initial memory and CPU states for initiating the execution of P in E. Let $m_{A,0}$ and $c_{A,0}$ be the same for the environment A containing the analyzer. Therefore, the traces of the program P in these two environments are defined as: $T(P, E) = (i_{E,0}, i_{E,1}, ..., i_{E,l_E})$, where $(i_{E,t}, m_{E,t}, c_{E,t}) \xrightarrow{\delta_{E,I_P}} (i_{E,t+1}, m_{E,t+1}, c_{E,t+1})$ for $0 \leq t < l_E$, and $T(P, A) = (i_{A,0}, i_{A,1}, ..., i_{A,l_A})$, where $(i_{A,t}, m_{A,t}, c_{A,t}) \xrightarrow{\delta_{A,I_P}}$ $(i_{A,t+1}, m_{A,t+1}, c_{A,t+1})$ for $0 \leq t < l_A$.

For the "only if" part of the proof, assume that P_A is transparent, we will prove by induction that $T(P, E) = T(P, A)$. The base case is trivial because the program starts execution at the same instruction i_0, so $i_{E,0} = i_{A,0} = i_0$. For the induction hypothesis, assuming $i_{E,t} = i_{A,t}$ and we need to prove that $i_{E,t+1} = i_{A,t+1}$. Since P_A is transparent, according to Definition 1, we have $d_p(A) = d_p(E) = 0$, and P will not try to change its intended execution path. Further, $d_P(A) = 0$ implies that the data/values in $m_{E,t}$, $c_{E,t}$, $m_{A,t}$, and $c_{A,t}$ that are visible to P, and hence relevant to the execution of P, must be the same (otherwise, $d_P(A) = 1$). That is, from P's point of view, the execution semantics in A and E is equivalent. Therefore, the next instruction in I_P executed in A has to be $i_{E,t+1}$ as in E, i.e., $i_{A,t+1} = i_{E,t+1}$. Using induction, we have $T(P, E) = T(P, A)$. For the "if" part of the proof, we assume that $T(P, E) = T(P, A)$ for any malware P, and we need to prove that the P_A is transparent. We prove by contradiction. Assume that P_A is not transparent. This means that according to Definition 1, we have $d_p(A) = 1$. Therefore, without loss of generality, P will alter its execution in A, which leads to the contradiction that $T(P, E) \neq T(P, A)$. Therefore, P_A is transparent. □

4.3.3 Requirements for Transparency

We use Definition 1 and Theorem 1 as guidelines to formulate the requirements on the design of a transparent malware analyzer. Our discussion here uses a generalization of several common hardware and operating system features such as privilege levels, virtual memory, and exception handling, which also covers other protection features provided by hardware virtualization. We first describe these features as parts of an extension to the basic program execution semantics introduced in Section 4.3.1, and then discuss the requirements for transparent malware analysis.

In order to achieve transparency (i.e., $d_P(A) = 0$ and hence $T(P, E) = T(P, A)$), the memory and CPU states visible to P and the instruction execution semantics need to be identical in both E and A. However, the presence of P_A and its analysis activities introduce changes to these entities.

Similar to information flow models in multi-level security systems [27], the notion of *privilege* is essential for reasoning about how to hide these changes from P. Suppose that there are n rings of privilege where 0 is the most privileged (or "highest") level and n is the least. Let the highest privilege level gained by the program P during execution in E be denoted by $\Pi_E(P)$. In order to represent virtual memory, suppose V denotes all possible virtual memory states viewed by instructions executed at a specific privilege level. The entire memory state M can then be defined as $M = V^n$, where each member $m \in M$ is an n-tuple of memory states, and $m[k] \in V$ is the state of the virtual memory at ring k. Virtual memory mapping can be expressed by functions $\mu^-_{E,r\to k}$, which maps memory of ring r to ring k and $\mu^+_{E,k\to r}$, which maps memory of ring k to ring r for $r > k$. By assuming that $\mu^-_{E,r\to k}$ and $\mu^+_{E,k\to r}$ are in $m[k]$, we can express how a higher privilege ring k code can control how a lower privilege ring r views its memory.

We use exceptions and exception handling to represent a broad range of system features such as all privileged instructions (e.g. system calls), I/O, memory content

or CPU register protection and access violations, and program and system faults. An exception occurs when an instruction execution requires services or data at a higher privilege level k than the current level r. A function $\phi_{E,r\to k}$ specifies the first instruction of the exception handler at ring k that handles the particular exception occurs at ring r.

The instruction execution semantics δ_E introduced in Section 4.3.1 can be extended to include two parts. The first is δ, the low level or *basic* instruction execution semantics that do not involve exceptions, and only deals with access to virtual memory and CPU registers (note that we consider I/O as exceptions). The second is $\delta_{E,\phi}$, the semantics that deal with exceptions (e.g., control transfers to and from exception handlers residing in privileged levels).

We now formulate the requirements of transparency.

1. Higher Privilege: We require that the analyzer P_A have higher privilege than the maximum privilege a malware P can gain. If the maximum privilege gained by P is $\pi = \Pi_A(P)$, then the analyzer should reside in privilege levels $k < \pi$. For any memory state $m \in M$, besides the code and data of P_A, the memory mapping functions $\mu^-_{A,\pi\to k}$ and $\mu^+_{A,k\to\pi}$ as well as the exception handler function $\phi_{A,\pi\to k}$ should also reside in $m[k]$. Proper isolation and protection can be achieved by ensuring $\mu^+_{A,k\to\pi}$ does not map any of these components to virtual memory state of $m[\pi]$.

2. No non-privileged side effects: This requirement states that if P_A induces side-effects, access to them should be privileged and through exception handlers at a higher privilege level(s) than P's. This ensures that any access to the changes in the memory, CPU registers, etc., can be intercepted using an exception handler that can hide these side-effects from P. Similarly, since P_A can have timing side-effects, instructions that can access any notion of time should be privileged as well.

3. Identical Basic Instruction Execution Semantics: Recall that the basic execution semantics do not involve any exception. This requirements states that basic

instruction execution semantics in A should be identical to that in E. From the second requirement above, the basic semantics do not involve any side-effects introduced by P_A (which is privileged and requires exception handling). Thus, the identical basic semantics, plus transparent exception handling (see requirement 4 below), guarantee that the same instruction has the same execution (and will lead to the same next instruction) in both A and E.

4. Transparent Exception Handling: Suppose that when the tth instruction is executed (in ring π), an exception occurs and the control is transfered to $\phi_{A,\pi\to k}$ in ring $k < \pi$. First, consider the case where there was no equivalent exception handler in environment E for the same instruction $i_{E,t}$ (i.e., $i_{E,t}$ was a basic instruction in E). In this case, handler code must first guarantee the third requirement above by executing $i_{A,t}$ with the same semantics as for $i_{E,t}$. Then, it has to guarantee that execution is returned to $i_{A,t+1}$ (the same as $i_{E,t+1}$) at the end of exception handling. In addition, the changes in $m_{A,t+1}$ and $c_{A,t+1}$ by the exception handler should only be privileged side-effects to fulfill the second requirement above. Second, if this exception handler replaces an original exception handler $\phi_{E,\pi\to k}$ in E, it needs to have identical changes made to $m_{A,t+1}$ and $c_{A,t+1}$ as $\phi_{E,\pi\to k}$ would make to $m_{E,t+1}$ and $c_{E,t+1}$ (e.g. results of system calls remain the same). The cases when the handlers involve timing measurements are addressed separately, as in fifth requirement below.

5. Identical Measurement of Time: This requirement states that the timing information received by the t'th instruction $i_{A,t}$ in $T(P, A)$ is the same as it were in $T(P, E)$. It is necessary to fulfill the transparency theorem because any changes in timing can be used to alter execution. For the malware P to view a continuous false view of time it is required that A maintain a *privileged* logical clock that is adjusted when exceptions (which include any access to the clock) are handled. Although the requirement of having identical measurement of time is very difficult to fulfill in practice, we can break it down to smaller requirements that may be easier to satisfy

in many cases. Suppose that the time spent in E to move from tth instruction to $(t+1)$th instruction is $\Delta_{E,t}$. We can define $\Delta_{E,t} = \Delta_{E,\delta,t} + \Delta_{E,\phi,t}$ where $\Delta_{E,\delta,t}$ is the time for basic instruction execution and $\Delta_{E,\phi,t}$ is the exception handling time. Similarly, $\Delta_{A,t} = \Delta_{A,\delta,t} + \Delta_{A,\phi,t}$. A needs to ensure $\Delta_{A,t} = \Delta_{E,t}$ by making some adjustments $\Delta'_{A,t}$, where $\Delta_{E,t} = \Delta_{A,t} - \Delta'_{A,t}$. Therefore, we have $\Delta'_{A,t} = \Delta_{A,\delta,t} + \Delta_{A,\phi,t} - \Delta_{E,\delta,t} - \Delta_{E,\phi,t}$. If the basic instruction execution requires the same amount of time in both A and E, we have $\Delta'_{A,t} = \Delta_{A,\phi,t} - \Delta_{E,\phi,t}$. Thus, when no exceptions occur for both E and A (i.e., both $\Delta_{A,\phi,t}$ and $\Delta_{E,\phi,t}$ are 0) no adjustment in time is required. When the an exception occurs for A but not for E (i.e., $\Delta_{E,\phi,t} = 0$), which is usually due to having privileged side-effects, $\Delta_{A,\phi,t}$, the time spent by the exception handler ϕ_A, has to be determined and negated. When both environments have exceptions, if ϕ_A is essentially ϕ_E plus some extra activities, then the extra time (i.e., $\Delta'_{A,t}$) can be measured and negated because the activities belonging to ϕ_E are executed and can be timed. However, if ϕ_A replaces ϕ_E (i.e., implements different activities), then it is very difficult to measure $\Delta'_{A,t}$ because ϕ_E is not executed.

4.3.4 Fulfilling the Requirements

We will now use the requirements presented in Section 4.3.3 to analyze the transparency achievable by various malware analysis approaches. In particular, we describe which transparency requirements the software virtualization and full system emulation based approaches cannot satisfy, and discuss how hardware virtualization extensions in the x86 architecture can overcome these limitations. The design and implementation of *Ether* are presented in Section 4.4.

Previous malware analysis approaches employ user level or kernel level counterparts residing in the host in which malware is analyzed; these include VAMPiRE, CWSandbox, and OmniUnpack. Since malware that use rootkit components can gain kernel level privileges, these approaches cannot satisfy the first requirement of

transparency.

Reduced privilege guest-based virtualization approaches (e.g., VMware [10] and VirtualPC [9] for x86) can fulfill the first requirement by emulating a few sensitive instructions in order to gain higher privilege over the OS kernel in the virtual machine. Therefore, the second requirement is partially satisfied by these approaches as they can remove certain memory and CPU side effects by providing a virtual view of memory. However, these approaches are not designed with transparency in mind, and the communication medium between the guest and host operating systems introduces unprivileged side effects. Moreover, instructions that can access time are not privileged, making these side effects visible in such systems through time measurement. In contrast, full system emulators (e.g. QEMU) emulate the entire low level instruction execution semantics δ to gain privilege over the guest OS. These approaches have privilege over all instructions executed, thereby fulfilling the second requirement.

An analyzer based on hardware virtualization extensions can likewise satisfy the first and second requirements. The first requirement is satisfied because the analyzer can reside in a domain more privileged than the guest. This privilege is enforced in hardware by the analyzer residing in *ring -1*, which has higher privilege than rings 0 to 3. In addition, the contents of the analyzer's domain are completely isolated through the use of shadow page tables. Hardware virtualization extensions not only enable basic memory protections, but also offer privileged access to sensitive CPU registers and instructions including instructions that access time, such as `RDTSC`. A malware analyzer based on these extensions can therefore intercept and hide these side effects from malware.

Neither reduced privilege guest-based approaches nor full system emulators can guarantee the third requirement. To elaborate, emulation-based approaches use low-level instruction execution semantics function δ' to simulate the entire low level execution semantics of δ. For reduced privilege guest-based approaches, δ' partially

simulates δ. The low level instruction execution semantics of both δ can be easily shown to be Turing complete. Likewise, δ' is also Turing complete. In addition, determining whether δ' is equivalent to δ requires determining whether all programs exhibit the same behavior under δ' and δ.

In automata theory, the above problem would be formally represented as the problem of determining whether the language of two Turing machines (L and L') are equal; it is otherwise known as the undecidable problem EQ_{TM} [134]. In practice, there have been attacks that detect full system emulator privilege guest-based approaches by exploiting incomplete emulation [57]. There is no way to guarantee the absence of such attacks, in the same way that software testing can only show the presence (and not absence) of bugs. In contrast, hardware virtualization extensions rely on the same hardware execution semantics δ, thereby guaranteeing that the third requirement is satisfied.

The fourth requirement is an analyzer design issue and can be satisfied by all approaches; it requires only careful design.

Finally, although emulators can have privileged access over instructions that can access the notion of time, it is non-trivial to provide a notion of time that is equivalent to the environment E. To elaborate, in emulators, almost all instructions have exception handlers managing their execution, and the identification of $\Delta_{E,t}$ is hard without having a real system execute these instructions in parallel. Moreover, the determination of $\Delta_{A,t}$ requires a cycle-count accurate execution simulator, which keeps track of the number of cycles required to execute an instruction in a real system.

In contrast, for hardware virtualization extensions-based approaches, the side effect on time is privileged because the instructions that access time (e.g., `RDTSC`) are privileged. As we describe next in Section 4.4, hardware virtualization extensions maintain a separate execution cycle count in the hypervisor, which allows an analyzer to adjust a cycle value before it is given to the guest. As such, while there still exist

complex situations that are hard to satisfy, hardware virtualization extension-based approaches go a long way in satisfying the fifth requirement.

4.4 Implementation

In this section we describe Ether's architecture, the low-level details of how it performs tracing, and the efforts necessary to ensure transparency in accordance with the Malware Analysis Theorem. In addition, we describe implementation challenges and current architectural limitations preventing maximal transparency.

4.4.1 Environment

To create Ether, we needed an analysis mechanism that was readily available to researchers and would allow for maximum transparency per the Malware Analysis Theorem. We deemed hardware virtualization extensions as the most appropriate, as they do not interfere with the original instruction stream I_P, the CPU registers C, the memory state M, the original exception handlers, as well as the original CPU transition function δ. In addition to this transparency, processors with such extensions are inexpensive and widely available.

Among available software which can utilize hardware virtualization extensions, we chose the Xen hypervisor version 3.1.0 as the base for implementing Ether. Xen was chosen because it is a mature product, it is open source, and it has existing communication mechanisms that could be leveraged in Ether's implementation. Finally, our work could also be incorporated into the numerous research projects currently supporting Xen.

Among hardware virtualization platforms we selected Intel VT due to the available documentation, our familiarity with Intel processors, and the availability of Intel-based hardware. Finally, as the selection of the target operating system must well represent actual software malware encounters in the wild, we chose Windows XP (Service Pack 2). Windows XP is the most common PC operating system in use

today and therefore a preferred target of modern malware.

4.4.1.1 A brief overview of Intel VT Hardware Virtualization Extensions

Intel VT Hardware virtualization extensions are a set of instructions added to Intel processors to facilitate the virtualization of the x86 instruction set. These instructions enable two new process modes, called VMX root mode and VMX non-root mode. The Xen hypervisor, and hence Ether, runs in VMX root mode. The domUs or guests, which we refer to as the *analysis targets*, run in VMX non-root mode.

Events called VM transitions change operation between the two modes. There are two different transitions, VMEntry and VMExit. A VMExit will transition from VMX non-root mode to VMX root mode, and a VMEntry will transition from VMX root mode to VMX non-root mode. Certain events in VMX non-root mode automatically cause a VMExit; these include certain exceptions, changes to the page directory entry pointer, and page faults, among others.

Ether obtains control from the analysis target on VMExits and performs a VMEntry when it chooses to resume execution of the analysis target. After a VMExit, but before the next VMEntry, the guest is in a completely dormant state.

4.4.2 Analyzer Architecture

The architecture of Ether consists of a hypervisor component and a userspace component running in domain 0. Ether's hypervisor component is responsible for detecting events in the analysis target. Currently, such events include system call execution, instruction execution, memory writes, and context switches.

Ether's userspace component acts as a controller regulating which processes and events in the guest should be monitored. This component also contains logic to derive semantic information from analyzed events, such as translating a system call number into system call name and displaying system call arugment content based on argument data type.

The analysis target consists of a Xen domU running Windows XP Service Pack 2. The only change we made to the default installation of Windows XP was disabling PAE and large memory pages. These modifications exist solely to make memory write detection easier in the initial Ether implementation and are not a limitation of our approach.

4.4.3 Using Intel VT Extensions for Malware Analysis

To present Ether as a full-featured malware analyzer we required that it be able to monitor the instructions executed by a guest process, any memory writes a guest process performs, and any system calls a guest process makes. We chose these low- and high-level operations due to their usefulness in malware analysis their ability to demonstrate the efficacy of Ether performing both coarse- and fine-grained tracing. The challenges to successful implementation included using a mechanism that was not intended for malware analysis (i.e., Intel VT) while maintaining the original level of transparency provided by hardware virtualization extensions. Given that Intel VT extensions do not provide explicit support for any of these monitoring activities we performed an in-depth investigation of Intel VT to create novel ways that fulfill our monitoring requirements. Our methods are described below.

4.4.3.1 Monitoring Instruction Execution

Instruction execution monitoring relies on the Ether's privilege over the analysis target in handling debug exceptions, and in guaranteeing a debug exception occurs after the execution of every instruction. Ether guarantees the occurence of a debug exception after every instruction by setting a flag called the trap flag in the analysis target. Upon handling a debug exception caused by the forced trap flag, Ether will once again set the trap flag for the next instruction, thereby inducing a debug exception after every instruction. Ultimate control of which exceptions reach the analyzed environment rests in Ether, so all induced debug exceptions are hidden from the

analysis target, In this manner, Ether executes the target process one instruction at a time while preventing it from detecting Ether's presence. Of note, this form of instruction stepping via the trap flag was first implemented in VAMPiRE, even though the VAMPiRE approach is in-guest and hence vulnerable to in-guest detection attacks.

4.4.3.2 Monitoring Memory Writes

Ether monitors memory writes by using shadow page tables and privilege over the guest in handling page faults. Ether induces a page fault at every attempted guest memory write, traps the fault, and, prevents it from reaching the analyzed environment. Faults occuring due to natural guest operation are forwarded to the analyzed environment. In this manner all analyzed environment memory write attempts are transparently intercepted.

Shadow page tables refer to the actual page tables the hardware uses for address translation, as the analyzed environment is never permitted to do its own translation. The hypervisor is responsible for synchronizing shadow page tables with the analyzed environment's page table contents, as well as ensuring the analyzed environment can only map memory explicitly allocated for it.

Ether causes page faults on write attempts by removing writeable permissions from shadow page table entries. When Ether detects a fault caused by itself, the Ether userspace component is notified of an attempted memory write, and the fault is then hidden from the analyzed environment. Faults resulting from normal guest operation are passed along. After notification, the faulting page is set to writeable in the shadow page table and the faulting instruction is re-executed. Upon completion of the instruction, all pages are once again marked read-only to detect any writes caused by the next instruction.

4.4.3.3 Monitoring System Call Execution

Ether's novel system call execution monitoring exploits features of the x86 fast system call entry mechanism to inform Ether of system calls executed by the analysis target. The system call interception mechanism uses a special register present on all modern x86 processors to cause a page fault at a chosen address during system call invocation. A fault at this address will then signal to Ether that a system call was executed in the analysis target.

To properly describe Ether's system call tracing technique, some background on the fast system call entry mechanism of x86 processors is required. This method is used for system calls on all Windows versions starting with XP, and on Linux kernels starting with 2.6. The fast system call entry mechanism uses the `SYSENTER` instruction to raise privilege and jump to a pre-defined address in the kernel. This address is stored in a special register called `SYSENTER_EIP_MSR`, access to which is only permitted from kernel mode. Whenever a userspace application requires system services, it specifies the service number and parameters in an implementation dependent manner, and then executes `SYSENTER`. `SYSENTER` changes the privilege mode to kernel mode, the stack pointer to the kernel mode stack, and the instruction pointer to the value in `SYSENTER_EIP_MSR`.

Ether sets the value of the analysis target's `SYSENTER_EIP _MSR` to a chosen value on a page guaranteed to be not present, and stores the original value in Ether's memory. Whenever the analyzed environment attempts system call execution, a fetch page fault will occur at the chosen address. When Ether encounters such a fault in the analyzed environment, it indicates a system call was attempted. The userspace component of Ether is notified of system call execution, and the instruction pointer of the analyzed environment is reset to the expected value of `SYSENTER_EIP_MSR`. Execution of the analyzed environment resumes as if `SYSENTER` jumped directly to the expected address, instead of our chosen address.

4.4.3.4 Limiting Scope to a Chosen Process

Ether gains control on every context switch by leveraging a feature of Intel VT that causes a VMExit every time the page directory entry pointer is accessed in the guest OS. As the page directory pointer must be updated every context switch to change address spaces, it is guaranteed that Ether will detect all guest context switches.

4.4.4 Maintaining Transparency

Despite making several modifications to the guest, Ether maintains transparency by ensuring such changes are undetectable. Other changes made by Ether, such as those to shadow page tables, are changes outside of the analyzed environment and therefore transparent.

4.4.4.1 Hiding the Trap Flag

Ether is able to conceal it has the set trap flag in the guest by intercepting the few instructions that can read this flag, and altering their behavior to hide the flag's presence. The only instruction which can directly read the presence of the trap flag is `PUSHF`. Ether intercepts this instruction and change its result to provide the environment-expected state of the flag. Besides `PUSHF`, the `INT` instruction reads the value of the flag indirectly. We monitor this instruction as well, and fix the flags value pushed on the stack to match the value expected by the analysis target.

At times the analysis target has set trap flag and expects to receive debug exceptions. We detect the setting of the trap flag by the `POPF` instruction, and when the analysis target expects the trap flag to be set, we forward debug exceptions along as appropriate. Any debug exceptions not caused by the trap flag are automatically forwarded to the analysis target as they are not caused by Ether.

4.4.4.2 Page Table Modifications

The page table modifications made by Ether are to shadow page tables, and not the page tables stored in the guest. Therefore, the guest is not aware that the shadow page tables exist, and hence cannot detect their modification or presence. Ether can determine which faults were caused by normal guest operation, and which faults are purposely it created. Any faults caused by normal operation are forwarded to the guest. Conversely, any faults caused by Ether are handled by Ether and never passed along.

4.4.4.3 SYSENTER_EIP_MSR

Ether mediates all access to the `SYSENTER_EIP_MSR` register and can therefore conceal any modifications of the register from the analyzed environment. Complete mediation is achieved because any access of the register automatically causes a VMExit. Ether saves any value the analysis target attempts to write into `SYSENTER_EIP_MSR`, and uses this value any time the guest-expected value of this register is required. Although technically modified, as observed from the analyzed environment the CPU state C is completely unchanged.

4.4.5 Potential Attacks

While theoretically resilient against in-guest detection attacks, current architectural restrictions make some of these attacks possible, and Ether is also vulnerable to a class of timing attacks using external timing sources.

4.4.5.1 Attack Classes

Classic in-guest detection attacks cannot detect Ether due to it being completely outside of the analyzed environment. Also, none of the few modifications Ether makes to the guest environment are unconditionally detectable. Ether is, however, vulnerable to a certain class of timing attacks, and in the current implementation, memory

hierarchy attacks. Detection methods and our mitigation of them are outlined below.

In-memory presence Traditional detection attacks which rely on detecting the presence of an analyzer in memory will always fail against Ether, as it has no in-guest memory presence in guest.

CPU Registers Ether hides the few changes it makes in CPU state from the analysis target so that it is unable to detect deviation from a native hardware environment.

Memory Protection Ether modifies only the shadow page tables, which are inaccessible to the analysis target. That is, the analysis target is unable to detect changes to shadow memory permissions. However, in the current implementation, Ether does indirectly modify the memory hierarchy (the cache and the TLB). This is due to an architectural limitation; further details appear in Section 4.4.6.

Privileged Instruction Handling Ether uses built-in hardware mechanisms to intercept only certain privileged instructions and exceptions and as necessary, forwards these exceptions to the guest. From the viewpoint of the guest, no handler is ever modified, and privileged instructions have the same effects as a native environment.

Instruction Emulation Ether does not emulate instructions; all instructions are executed on the actual processor. Therefore, Ether does not suffer from emulation inaccuracies inherent in full system x86 emulators; the transition function δ remains unmodified.

Timing Attacks As described in Section 4.3.3, there are two fundamental issues with avoiding timing attacks: controlling queries about time, and answering those queries with the expected time value. Ether controls the in-guest view of the `RDTSC` instruction, the APIC timer, the 8254 timer chip, as well as any periodic time-based interrupts and other guest time queries. Section 4.3.3 also outlines the requirements for a correct reply to a guest time query. This correct reply, $\Delta_{E,t}$, is the wall-clock

time when Ether is present, $\Delta_{A,t}$, reduced by $\Delta'_{A,t}$, the amount of overhead added by Ether's presence. $\Delta'_{A,t}$ is equal to the time spent in the analyzer minus the time spent in the guest exception handler ($\Delta'_{A,t} = \Delta_{A,\phi,t}$ - $\Delta_{E,\phi,t}$). As Ether always calls the native exception handlers, if any, for any exceptions the guest must process, $\Delta_{E,\phi,t}$ is always zero. The remaining term, $\Delta_{A,\phi,t}$ consists of the time spent switching privilege to Ether, the time spent in Ether's handler, and the time to switch to the original guest handler. The time spent in the handler is easily measured using the wall-clock. The times for privilege changes can be calculated empirically. Therefore, Ether would be able to adjust the time returned to the guest by the amount of overhead, and remain untraceable by timing attacks which rely on host-based time sources.

Additionally, the Intel VT architecture has a special field, called the `TSC_OFFSET`, which the processor will automoatically add to any queries of the time stamp counter. This value must be set to the calculated value of $\Delta'_{A,t}$. Any other clock sources, all of which are under control of Ether, can be adjusted accordingly.

As Ether can only mitigate in-guest timing detection attacks, a class of detection attacks utilizing time from external inputs is still possible. However, preventing such attacks would require detecting covert channels over a shared medium, which is known to be undecidable [47].

4.4.6 Architectural Limitations

Intel VT suffers from some architectural limitations which may allow Ether to be detected under circumstances. Different hardware virtualization extensions exist which do not suffer from such limitations.

Intel VT suffers from two main flaws which allow the current implementation to be detected by observing implicit changes to the memory hierarchy. The first flaw is that Intel VT induces a TLB flush on every VMExit. A detection method such as the one proposed by [matasano guy] can detect this implicit change. A hypervisor could run

in non-paged mode with caches off in an effort to avoid using the extended memory hierarchy at all. Intel VT's second flaw is that paging mode must be turned on before entering VMX Root Mode. AMD's competing virtualization solution, AMD-V, does not suffer from either of these limitations. A hypervisor which did not use paging and ran with caches off would be undetectable by such memory hierarchy detection methods.

4.5 Summary

The Ether approach of using hardware virtualization for malware analysis provides a transparent multi-grained malware analysis framework. The tools built on Ether, EtherUnpack and EtherTrace, are more effective than previous approaches in unpacking and system call tracing, respectively. Ether also achieves a level of transparency not matched by emulation based or in-guest analysis frameworks. Due to Ether's use of hardware assisted virtualization, Ether avoids an in-guest presence as well as emulation inaccuracies inherent to system emulators.

Modern malware is obfuscated with packers which are increasingly targeting virtualization and emulation based environments. Samples detecting VMWare and QEMU have been found in the wild, even among such common malware as Storm. Packers such as Themida now provide specific options to detect virtualized environments and halt execution. Emulators can never be guaranteed to perfectly emulate a physical processor; malware can always utilize detection based on imperfect emulation. As such obfuscated and emulation-resistant malware becomes more prevalent, Ether will become increasingly more valuable in targeting this evolving threat.

CHAPTER V

CONDITIONAL CODE OBFUSCATION

5.1 Motivation

While static analysis suffers from being vulnerable to a vast array of obfuscations, dynamic analysis techniques can provide a better tolerance against these attacks. In order to defend against run-time anti-analysis tricks that target dynamic analysis techniques, such as debugger detection, virtual machine detection, emulator detection, etc., we proposed the transparent malware analysis model and the Ether framework in the previous chapter. The problem with straight forward dynamic anlaysis techniques is that only a single execution path is seen during each execution, and several branches of execution may not be explored. Malware authors exploit this drawback by employing trigger-based behavior, or malicious activities that are activated when a specific condition is met. Without these conditions being satisfied, the malicious activities become hidden to the analysis.

Recent malware analyzers provide a powerful way to discover trigger based malicious behavior in arbitrary malicious programs. Moser et al. proposed a scheme [101] that explores multiple paths during execution of a malware. After exploring a branch, their technique resumes execution from a previously saved state and takes the alternate branch by inverting the condition and solving constraints to modify related memory variables in a consistent manner. Other recently proposed approaches can make informed path selection [31], discover inputs that take a specific path [32, 38, 45] or force execution along different paths [162]. We call all such approaches *input-oblivious analyzers* because they do not utilize any source of information about inputs other than the program being analyzed.

Our goal is to anticipate attacks against the state-of-the-art malware analyzers in order to develop more effective analysis techniques. We present a simple, automated and transparent obfuscation against powerful input oblivious analyzers. We show that *it is possible to automatically conceal trigger-based malicious behavior of existing malware from any static or dynamic input-oblivious analyzer by an automatically applicable obfuscation scheme based on static analysis.* Our attack falls into the *input-based obfuscation* class, which we mentioned in Chapter 2.

Our scheme, which we call *conditional code obfuscation*, relies on the principles of secure triggers [59]. First, we identify and transform specific branch conditions that rely on inputs by incorporating one-way hash functions in such a way that it is hard to identify the values of variables for which the conditions are satisfied. Second, the *conditional code*, which is the code executed when these conditions are satisfied, is identified and encrypted with a key that is derived from the value that satisfies the condition. As a result, input oblivious analyzers can no longer feasibly determine the values that satisfy the condition and consequently the key to unveil the conditional code. Our approach utilizes several static analysis techniques, including control dependence analysis, and incorporates both source-code and binary analysis to automate the entire process of transforming malware source programs to their obfuscated binary forms.

In order to show that conditional code obfuscation is a realistic threat, we have developed a compiler-level tool that applies the obfuscation to malware programs written in C/C++. Our prototype implementation generates obfuscated compiled ELF binaries for Linux. Since the malware authors will be the ones applying this technique, the assumption of having the source code available is realistic. We have tested our system by applying it on several real malware programs and then evaluated its effectiveness in concealing trigger based malicious code. In our experiments on 7 different malware programs containing 92 malicious triggers, our tool successfully

obfuscated and concealed the entire code that implemented 87 of them.

We analyze the strengths and weaknesses of our obfuscation. Although the keys are effectively removed from the program, the encryption is still susceptible to brute force and dictionary attacks. We provide a method to measure the strength of particular applications of the obfuscation against such attacks. To understand the possible threats our proposed obfuscation scheme poses, we discuss how malware authors may manually modify their code in different ways to take advantage of the obfuscation technique. Finally, we provide insight into possible ways of defeating the obfuscation scheme, including more informed key search attacks and the incorporation of input-domain information in existing analyzers.

The goal of our obfuscation is to hide malicious behavior from malware analyzers that extract behavior. For these analyzers, the usual assumption is that the program being analyzed is already suspicious. Nevertheless, malware authors wish to develop code that is not easily detected. Naïve usage of this obfuscation may actually improve malware detection because of the particular way in which hash functions and decryption routines are used. However, since attackers can add existing polymorphic or metamorphic obfuscation techniques on top of our technique, a detector should be able to detect such malware at best with the same efficacy as polymorphic malware detection.

5.2 Previous Work

The problem of discovering trigger-based malicious behaviors has been addressed by recent research. Some techniques have used symbolic execution to derive predicates leading to specific execution paths. In order to identify time-bombs in code, [45] varies time in a virtual machine and uses symbolic execution to identify predicates or conditions in a program that depend on time. Another symbolic execution based method of detecting trigger based behavior is presented in [32]. Brumley et al.'s

Bitscope [31] uses static analysis and symbolic execution to understand behavior of malware binaries and is capable of identifying trigger-based behavior as well. Our obfuscation makes it hard for symbolic constraints containing cryptographic one-way hash functions to be solved.

Approaches have been proposed that identify trigger-based behavior by exploring paths during execution. Moser et al.'s multi-path exploration approach [101] was the first such approach. The technique can comprehensively discover almost all conditional code in a malware with sufficient execution time. The system uses QEMU and dynamic tainting to identify conditions and construct path constraints that depend on inputs coming from interesting system calls. Once a conditional branch is reached, the approach attempts execution on both of the branches after consistently changing memory variables by solving the constraints. Another approach is presented in [162] that forces execution along different paths disregarding consistent memory updates. The approach has been shown to be useful for rootkits written as Windows kernel drivers.

Techniques that can impede analyzers capable of identifying trigger-based behavior need to conceal conditions and the code blocks that are used to implement these behaviors. An example of obfuscated conditional branches in malware was seen in the early 90s in the Cheeba [67] virus, which searched for a specific file by comparing the hash of the name of each file with a hard-coded hash of the file name being searched. In the literature, the idea of using environment generated keys for encryption was introduced in [120]. The work on secure triggers [59] considers a whitehat scenario and presents the principles of constructing protocols for software developers to have inputs coming into a program to decrypt parts of the code. In the malware scenario, the research idea of using environment generated keys for encryption was presented as the Bradley virus [58]. However, such techniques have not become a practical threat

because identification of such keys and incorporation of the encryption technique requires a malware to be manually designed and implemented around this obfuscation. Our work shows that encrypting parts of the malware code using keys generated from inputs can be automatically applied on existing malware without any human effort, showing its efficacy as a wide-spread threat.

The use of polymorphic engines in viruses is one of the earliest [136] obfuscation techniques used in malware to evade detection. Over the years, various methods of polymorphic and metamorphic techniques have appeared in the wild [143]. In order to detect polymorphic malware, antivirus software use emulation or create signatures to detect the polymorphic decryptors. In [40], obfuscation techniques such as garbage insertion, code transposition, register reassignment, and instruction substitution were shown to successfully disrupt detection of several commercial antivirus tools.

Besides malware detection approaches [41, 84, 79], recent research has focused on creating techniques that automate malware analysis. These systems automatically provide comprehensive information about the behavior, run-time actions, capabilities, and controlling mechanisms of malware samples with little human effort. A variety of obfuscation techniques [33, 34, 35, 166] have been presented that can impede such analyzers that are based on static analysis approach. Executable protectors and packers [132] are widely used by malware authors to make reverse engineering or analysis of their code very hard. As with polymorphic code, packers obfuscate the code by encrypting or compressing the binary and adding an unpacking routine, which reverses the operation during execution. Tools [122, 76, 96] have been presented that are able to unpack a large fraction of such programs to aid static analysis. However, by utilizing our obfuscation before packing, malware authors are capable of concealing code implementing triggered behavior from static analyzers even after unpacking is performed.

```
cmd  = get_command(sock);
if (strcmp(cmd,"startkeylogger") == 0)
{
    log_keys();
}
```

```
n = get_day_of_month();
if ((n > 10) && (n < 20))
{
    attack();
}
```

Figure 6: Two conditional code snippets.

Dynamic analysis approach has been more attractive for automated malware behavior analyzers. This is because performing analysis of code that is executing overcomes obfuscations that impede static analysis, including packed code. Since most dynamic analysis of malware involves debuggers [49], safe virtual machine execution environments or emulators, malware programs use various anti-debugging [39] and anti-analysis techniques to detect side-effects in the execution environment and evade analysis. There has been research on stealth analysis frameworks such as Cobra [146], which places stealth hooks in the code to aid analysis while remaining hidden from the executing malware. In order to automate analysis, dynamic tools such as CWSandbox [36], TTAnalyze [24] or the Norman Sandbox [11] automatically record the actions performed by an executing malware. However, since such tools can only view a single execution path, trigger-based behavior may be missed. These tools have been superseded by the recent approaches that can identify and extract trigger-based behavior, which we have presented earlier in this section.

5.3 Conditional Code Obfuscation

Malware programs routinely employ various kinds of trigger based events. The most common examples are bots [48], which wait for commands from the botmaster via a command and control mechanism. Some keyloggers [141] log keys from application windows containing certain keywords. Timebombs [98] are malicious code executed at a specific time. Various anti-debugging [39] or anti-analysis tricks detect side-effects in the executing environment caused by analyzers and divert program execution when present. The problem for the malware writer is that the checks inside the program that

are performed on the inputs give away information about what values are expected. For example, the commands a bot supports are usually contained inside the program as strings. More generally, for any trigger based behavior, the conditions recognizing the trigger reveal information about the inputs required to activate the behavior.

The basis of our obfuscation scheme is intuitive. By replacing input-checking conditions with equivalent ones that recognize the inputs without revealing information about them, the inputs can become secrets that the input-oblivious analyzer can no longer discover. Such secrets can then be used as keys to encrypt code. Since the modified conditions are satisfied only when the inputs are sent to the program, the code blocks that are conditionally executed can be encrypted. In other words, our scheme encrypts conditional code with a key that is removed from the program, but is evident when the modified condition is satisfied. Automatically carrying out this transformation as a general obfuscation scheme involves several subtle challenges.

We provide a high-level overview of our obfuscation with program examples in Section 5.3.1. The general mechanism is defined in Section 5.3.2. The program analysis algorithms and transformations required are described in Section 5.3.3. Section 5.3.4 describes the consequences of our scheme on existing malware analysis approaches. Section 5.3.5 discusses possible brute-force attacks on our obfuscation technique.

5.3.1 Overview

Figure 6 shows snippets of two programs that have conditionally executed code. The first program snippet calls a function that starts logging keystrokes after receiving the command "startkeylogger". The second example starts an attack only if the day of month is between the 11th and the 19th. In both the programs, the expected input can be easily discovered by analyzing the code.

We use cryptographic hash functions to hide information. For the first example, we can modify the condition to compare the computed the hard-coded hash of the

string in `cmd` with the hash value of the string "startkeylogger" (Figure 7). The command string "startkeylogger" becomes a secret that an input oblivious analyzer cannot know. This secret can be used as the key to encrypt the conditional code block and the entire function `log_keys()`. Notice that when the expected command is received, the execution enters the `if` block and the encrypted block is correctly decrypted and executed.

```
cmd  = get_command(sock);
if (hash(cmd) == H)
    /* here, H = hash("startkeylogger") */
{
    decrypt_function(encr_log_keys, cmd);
    encr_log_keys();  /* encrypted log_keys */
}
```

Figure 7: Obfuscated example snippet.

In the second example of Figure 6, cryptographic hashes of the operands do not provide a condition equivalent to the original. Moreover, since several values of the variable n satisfy the condition, it is problematic to use them as keys for encryption and decryption.

We define *candidate* conditions as those suitable for our obfuscation. A candidate needs three properties. First, the ordering relation between the pre-images of the hash function must be maintained in the images. Second, there should be a unique key derived from the condition when it is satisfied. Third, the condition must contain an operand that has a statically determinable constant value.

Given these requirements above, operators that check equality of two data values are suitable candidates. Hence, conditions having '==', `strcmp`, `strncmp`, `memcmp`, and similar operators can be obfuscated with our mechanism.

5.3.2 General Mechanism

We now formally define the general method of our conditional code obfuscation scheme. Without loss of generality, we assume that any candidate condition is equivalent to the simple condition "$X == c$" where the operand c has a statically determinable constant value and X is a variable. Also, suppose that a code block B is executed when this condition is satisfied. Figure 8 shows the program transformation required for the obfuscation. The cryptographic hash function is denoted by $Hash$ and the symmetric encryption and decryption routines are $Encr$ and $Decr$, respectively.

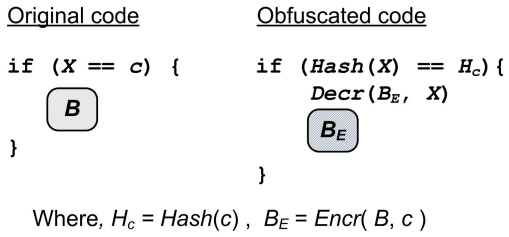


Figure 8: General obfuscation mechanism.

The obfuscated condition is "$Hash(X) == H_c$" where $H_c = Hash(c)$. The *pre-image resistance* property of the function $Hash$ implies that it is infeasible to find c given H_c. This ensures that it is hard to reverse the hash function. In addition, because of the *second pre-image resistance* property, it is hard to find another c' for which $Hash(c') = H_c$. Although this property does not strengthen the obfuscation, it is required to make the obfuscated condition semantically equivalent to the original, ensuring the correctness of the program.

The block B is encrypted with c as the key. Let B_E be the encrypted block where $B_E = Encr(B, c)$. Code is inserted immediately before B_E to decrypt it with the key contained in variable X. Since $Hash(X) = H_c$ implies $X = c$, when the obfuscated condition is satisfied, the original code block is found, i.e. $B = Decr(B_E, c)$ and the program execution is equivalent to the original. However, a malware analyzer

can recover the conditional code B only by watching for the attacker to trigger this behavior, by guessing the correct input, or by cracking the cryptographic operations.

5.3.3 Automation using Static Analysis

In order to apply the general mechanism presented in the previous section automatically on a program, we utilized several known algorithms in static analysis. In this section, we describe how these program analysis techniques were used.

5.3.3.1 Finding Conditional Code

In order to identify conditional code suitable for obfuscation, we first identify candidate conditions in a program. Let $\mathcal{F}$ be the set of all functions or procedures and $\mathcal{B}$ be the set of all basic blocks in the program that we analyze. For each function $F_i \in \mathcal{F}$ in the program, we construct a *control flow graph* (CFG) $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i)$ in the program where $\mathcal{V}_i \subseteq \mathcal{B}$ is the set of basic blocks in F_i and $\mathcal{E}_i$ is the set of edges in the CFG representing control-flow between basic blocks. We then identify basic blocks having conditional branches, which have two outgoing edges. Since we are not interested in conditions used in loops, we employ loop analysis to identify such conditions and discard them. From the remaining conditional branches, we select candidate conditions as the ones containing equality operators as described in Section 5.3.1. Let $\mathcal{C}_i \subseteq \mathcal{V}_i$ be the set of blocks containing candidate conditions for each function F_i.

After candidate conditions are identified in a program, the next step is to find corresponding conditional code blocks. As described earlier, conditional code is the code that gets executed when a condition is satisfied. It may include some basic blocks from the same function the condition resides in and some other functions. Since a basic block can contain at most one conditional branch instruction, by a condition, we refer to the basic block that contains it. We use the mapping $CCode : \mathcal{B} \rightarrow (\mathcal{B} \cup \mathcal{F})^*$ to represent conditional code for any condition.

In order to determine conditional code, we first use *control dependence analysis* [15,

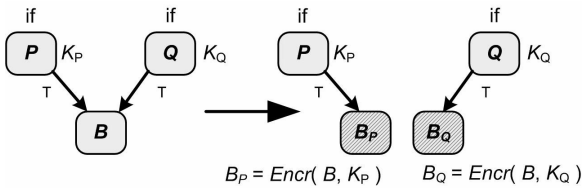


Figure 9: Duplicating conditional code.

55] at the intra-procedural level. A block Q is control dependent on another block P if the outcome of P determines the reachability of Q. More precisely, if one outcome of P always executes Q, but the other outcome may not necessarily reach Q, then Q is control dependent on P. Using the standard algorithm for identifying control dependence, we build a *control dependence graph* (CDG) for each function, where each condition block and their outcome has edges to blocks that are control dependent on it. For each candidate condition, we find the set of blocks that are control dependent on its true outcome. Therefore, if a true outcome of a candidate condition $C \in \mathcal{C}_i$ of function F_i has an edge to a block $B \in \mathcal{V}_i$, then $B \in CCode(C)$.

Conditional code blocks may call other functions. To take them into account, we determine reachability in the inter-procedural CFG. If there exists a call to a function F from a conditional code block $B \in CCode(C)$ of some candidate condition C, we consider $F \in CCode(C)$ which means that we consider all the code in the function F as conditional code of C as well. Now, for every block in $F \in CCode(C)$, we find calls to other functions and include them in $CCode(C)$. This step is performed repeatedly until all reachable functions are included. We used this approach instead of inter-procedural control-dependence analysis [133] because it allows us to obfuscate functions that are not control-dependent on a candidate condition but can be reached only from that condition.

A candidate condition may be contained in a block that is conditional code of another candidate condition. The next step is to eliminate these cases by making a function or block conditional code of only the closest candidate condition that can

reach it. For any block $B \in CCode(C)$, if $B \in CCode(C')$ where $C \neq C'$ and $C \in CCode(C')$ then we remove B from $CCode(C')$. We perform the same operation for functions.

Blocks and functions can be obfuscated when they are conditional code of candidate conditions only. If they are reachable by non-candidate conditions, then we cannot obfuscate them. When obfuscations are to be applied, they are applied in an iterative manner, starting with the candidate condition that have no other candidate conditions depending on it. The basic blocks and functions that are conditional code of these conditions are obfuscated first. In the next iteration, candidate conditions with no unobfuscated candidate conditions depending on it are obfuscated. The iterative process continues until all candidate conditions are obfuscated.

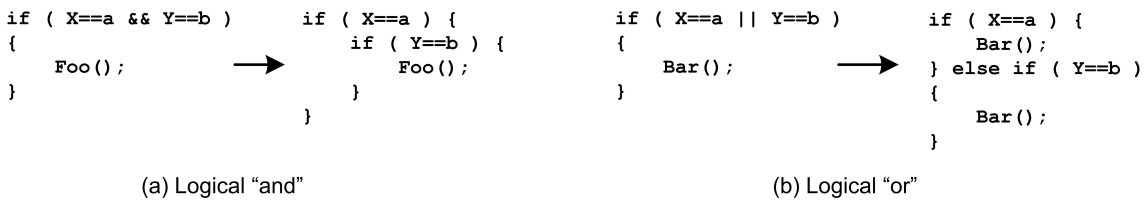


Figure 10: Compound condition simplification.

We use a conservative approach to identify conditional code when the CFG is incomplete. If code pointers are used whose targets are not statically resolvable, we do not encrypt code blocks that are potential targets and any other code blocks that are reachable from them. Otherwise, the encryption can crash the program. Fortunately, type information frequently allows us to limit the set of functions or blocks that are probable targets of a code pointer.

5.3.3.2 Handling Common Conditional Code

A single block or a function may be conditional code of more than one candidate condition that are not conditional code of each other. For example, a bot program may contain a common function that is called after receiving multiple different commands.

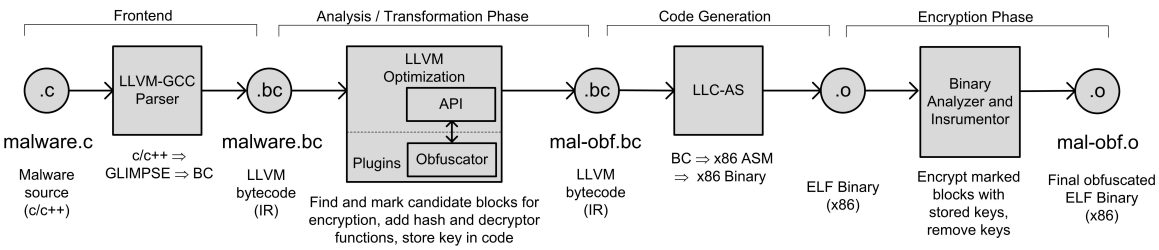


Figure 11: The architecture of our automated conditional code obfuscation system.

If a block B is conditional code of two candidate conditions P and Q, where $P \notin CCode(Q)$ and $Q \notin CCode(P)$ then B can be reached via two different candidate conditions and cannot be encrypted with the same key. As shown in Figure 9, we solve this problem by duplicating the code and encrypting it separately for each candidate condition.

5.3.3.3 Simplifying Compound Constructs

Logical operators such as `&&` or `||` combine more than one simple condition. To apply our obfuscation, compound conditions must be first broken into semantically equivalent but simplified conditions. However, parts of a compound condition may or may not be candidates for obfuscation, making the compound condition unsuitable for obfuscation.

Logical `and` operators (`&&`) can be written as nested `if` statements containing the operand conditions and the conditional block in the innermost block (Figure 10(a)). Since both the simple conditions must be satisfied to execute conditional code, the code can be obfuscated if at least one of them is a candidate condition.

Logical `or` operators (`||`) can be obfuscated in two ways. Since either of the conditions may execute conditional code, the conditional code may be encrypted with a single key and placed in two blocks that are individually obfuscated with the simple conditions. Another simple way is to duplicate the conditional code and use `if...else if` constructs (Figure 10(b)). Note that if either one of the two conditions is not a candidate for obfuscation, then the conditional code will remain revealed.

Concealing the other copy does not gain protection. Although it is not possible to determine that the concealed code is equivalent to the revealed one, the revealed code gives away the behavior that was intended to be hidden from an analyzer.

To introduce more candidate conditions in C/C++ programs, we convert `switch...case` constructs into several `if` blocks, each containing a condition using an equality operator. Every case except the *default* becomes a candidate for obfuscation. Complications arise when a code block under a switch case falls through to another. In such cases, code of the block in which control-flow falls through can be duplicated and contained in the earlier switch case code block, and then the standard approach that we described can be applied.

5.3.4 Consequences to Existing Analyzers

Our obfuscation can thwart different classes of techniques used by existing malware analyzers. The analysis refers to the notations presented in Section 5.3.2.

1. **Path exploration and input discovery:** Various analysis techniques have been proposed that can explore paths in a program to identify trigger based behavior. Moser et al.'s dynamic analysis based approach [101] explores multiple paths during execution by repeatedly restoring earlier saved program states and solving constructed path constraints in order to find a consistent set of values of in-memory variables that satisfy conditions leading to different paths. They use dynamic taint analysis on inputs from system calls to construct linear constraints representing dependencies among memory variables. After our obfuscation is applied, the constraint added to the system is "$Hash(X) == H_c$", which is a non-linear function. Therefore, our obfuscation makes it hard for such a multi-path exploring approach to feasibly find value assignments to variables in the programs's memory to proceed towards the obfuscated path.

2. **Discovering inputs:** A similar effect can be seen for approaches that discover inputs from a program that executes it along a specific path. EXE [38] uses mixed symbolic and concrete execution to create constraints that relate inputs to variables in memory. It uses its own efficient constraint solver called STP, which supports all arithmetic, logical, bitwise, and relational operators found in C (including non-linear operations). Cryptographic hash functions are designed to be computationally infeasible to reverse. Even with a powerful solver like STP, it is infeasible to generate and solve the constraints that represent the complete set of operations required to reverse such a function.

3. **Forcing execution:** A malware analyzer may force execution along a specific path without finding a consistent set of values for all variables in memory [162], hoping to see some malicious behavior before the program crashes. Suppose that the analyzer forces the malware program to follow the obfuscated branch that would originally execute B without finding the key c. Assuming X has a value c' where $c' \neq c$, the decrypted block then becomes $B' = Decr(B_E, c') \neq B$. In practical situations, subsequent execution of B' should cause the program to eventually crash without revealing any behavior of the original block B.

4. **Static analysis:** The utilization of hash functions alone cannot impede approaches utilizing pure static analysis or hybrid methods [31] because behavior can be extracted by analyzing the code without requiring constraint solving. However, our utilization of encryption conceals the behavior in the encrypted blocks B_E that can only be decrypted by the key c, which is no longer present in the program.

5.3.5 Brute Force and Dictionary Attacks

Although existing input-oblivous techniques can be thwarted by our obfuscation, analyzers that are aware of the obfuscation may attempt brute-force attacks on the

keys used for encryption. First, the hash function $Hash()$ being used in the malware needs to be extracted by analyzing the code. Then, a malware analyst may try to identify c by computing $Hash(X)$ for all suitable values of X and searching for a value satisfying $Hash(X) = H_c$. The strength of the obfuscation applied to the condition can therefore be measured by the size of the minimal set of suitable values of X.

Let $Domain(X)$ denote the set of all possible values that X may take during execution. If τ is the time taken to test a single value of X or the hash computation time, then the brute force attempt will take $|Domain(X)|\tau$ time. Finding the set $Domain(X)$ is not straightforward. In most cases, only the size of X may be known. If X is n bits in length, then the brute force attack requires $2^n\tau$ time. The possibility of using a pre-computed hash table to reduce search time can be thwarted by using a nonce with the data before computing the hash. Moreover, different nonce values for different conditions can be used to make the computed hash for one condition not useful for another.

Existing program analysis techniques such as data-flow analysis or symbolic execution may provide a smaller $Domain(X)$ in some cases, enabling a dictionary attack. Section 5.6 discusses more about attacks on our obfuscation, including automated techniques that can be incorporated in current analyzers.

5.4 Implementation Approach

In this section, we present the design choices and implementation of the compiler-level tool that we developed to demonstrate the automated conditional code obfuscation on malware programs. Our primary design challenge was to select the appropriate level of code on which to work on. Performing encryption at a level higher than binary code would cause un-executable code to be generated after decryption at run-time. On the other hand, essential high-level information such as data types require analysis at a higher level. Our system, therefore, works at both the intermediate code and the

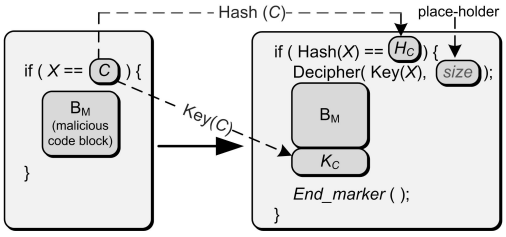


Figure 12: Analysis phase (performed on IR).

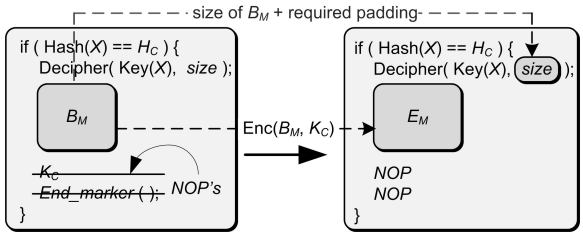


Figure 13: Encryption Phase (performed on binary).

binary level.

We use the LLVM [88] compiler infrastructure and the DynInst [3] binary analysis and instrumentation system. The LLVM framework is an extensible program optimization platform providing an API to analyze and modify its own RISC-like intermediate code representation. It then emits binary code for various processor families. Most of the heavy-duty analysis and code instrumentation is done using the help of LLVM. The DynInst tool is a C/C++ based binary analysis and instrumentation framework, which we use for binary rewriting.

We implemented our prototype for the x86 architecture and the Linux OS. We targeted the Linux platform primarily to create a working system to showcase the capability of the proposed obfuscation scheme without making it a widely applicable tool for malware authors. However, the architecture of our obfuscation tool is general enough to be portable to the Windows OS.

The architecture of our system is presented in Figure 11. Our system takes as input a malware source program written in C/C++ and generates an obfuscated binary in the Linux ELF format. The transformation is done in four phases. The phases are (1) the Frontend Code Parsing Phase, (2) the Analysis/Transformation Phase, (3) the Code Generation Phase, and (4) the Encryption Phase.

In the first phase LLVM's GCC-based parser converts C/C++ source programs to LLVM's intermediate representation. Next, in the analysis and transformation phase, the bulk of the obfuscation except for the actual encryption process is carried

out on the intermediate code. In this phase, candidate conditions are identified and obfuscated, conditional code blocks are instrumented with decipher and marker routines, and the key required for encryption is stored as part of the code. Section 5.4.1 describes these steps in details. In the third phase, we use the static compilation and linking back end of LLVM to convert the LLVM intermediate representation (IR) to x86 assembly from which we generate an x86 ELF binary. In the final phase, which is described in Section 5.4.2, our DynInst based binary modification tool encrypts marked code blocks to complete the obfuscation process. We describe in Section 5.4.3 how the decryption of code takes place during run-time.

5.4.1 Analysis and Transformation Phase

The analysis and transformation was implemented as an LLVM plugin, which is loaded by the LLVM optimization module. The steps taken are illustrated in Figure 12 and described below.

5.4.1.1 Candidate Condition Replacement

We followed the method described in section 5.3.3 to identify candidate conditions and their conditional code. As shown in Figure 12, the transformed code calls the hash function with the variable used in the condition as the argument. We use SHA-256 in our implementation as the hash function. The replaced condition compares the result with the hard coded hash value of the constant. In other words, the condition "$X == c$" is replaced with "$Hash(X) == H_c$", where $H_c = Hash(c)$. Depending on the data type of X, calls are placed to different versions of the hash function. For length constrained string functions, the prefixes of the strings are taken. A special wrapper function is used for `strstr`, which computes the hash of every sub-string of X and compares with H_c.

5.4.1.2 Decipher Routine

In our implementation, we selected AES with 256-bit keys as the encryption algorithm. Constants in the conditions are not directly used as keys because of the varying type and length. A key generation function $Key(X)$ is used to produce 256-bit keys. We describe the function in more details in the next section.

We use the LLVM API to place a call immediately before the original conditional code to a separate decipher function that can be dynamically or statically linked to the malware program. Figure 12 illustrates this for a basic block. For a function, the call to the decipher routine is inserted as the first statement in the function body. The *Decipher* routine takes two arguments. The first is a dynamically computed key $Key(X)$, which is based on the variable X in the condition. The second is the length of the code to be decrypted by the decipher routine. When calling a function that is to be obfuscated, these two arguments are added to the list of arguments for that function. This allows them to be passed not only to decipher routine called in that function body, but also to other obfuscated functions that the function may call. At this stage in the obfuscation scheme, this length is not known because the final generated code size will vary from the intermediate code. We keep a place holder so that the actual value can be placed during binary analysis.

5.4.1.3 Decryption Key and Markers

The key generation function Key uses a SHA-256 cryptographic hash to generate a fixed length key from varying length data. However, the system would break if we used the same hash function as used in the condition. The reason is that if the key $Key(c) = Hash(c) = H_c$, then the stored hash H_c in the condition can be used to decrypt the code blocks. Therefore, we use $Key(X) = Hash(X|N)$, where N is a nonce. This ensures that the encryption key $K_c = Key(c) \neq H_c$, where c is the constant in the condition. At this stage, the code block to be encrypted is not

modified. Immediately following this block, we place the encryption key K_c.

During intermediate code analysis, it is not possible to foresee the exact location of the corresponding code in the resulting binary file. Therefore, we place markers in the code, which are later identified using a binary analyzer in order to perform encryption. The function call to *Decipher* works as a beginning marker and we place a dummy function call *End_marker*() after the encryption key. We use function calls as markers because the LLVM optimization removes other unnecessary instructions from the instruction stream. This type of placement of the key and markers have no abnormal consequences on the execution of the program because it is identified during binary analysis and removed at the final stage of obfuscation.

5.4.2 Encryption Phase

With the help of DynInst, our tool analyzes the ELF binary output by LLC. In order to improve code identification, we ensure that symbol information is intact in the analyzed binary.

Figure 13 illustrates the steps carried out in this phase. At this stage, our tool identifies code blocks needing encryption by searching for calls to the marker functions *Decipher*() and *End_marker*(). When such blocks are found, it extracts the encryption key K_c from the code and then removes the key and the call to the *End_marker* function by replacing them with x86 `NOP` instructions. It then calculates the size of the encrypted block. Since AES is a block cipher, we make the size a multiple of 32 bytes. This can always be done because the place for the key in the code leaves enough `NOP`s at the end of the code block needing encryption. We place the size as the argument to the call to *Decipher*, and then encrypt the block with the key K_c.

The nested conditional code blocks must be managed in a different way. We recursively search for the innermost nested block to encrypt, and perform encryption starting from the innermost one to the outermost one. Since our method of encrypting

the code block does not require extra space beyond what is already reserved, our tool does not need to perform code movement in the binary.

5.4.3 Run-time Decryption Process

The *Decipher* function performs run-time decryption of the encrypted blocks. Notice that the location of the block that needs to be decrypted is not sent to this function. When the *Decipher* function is called the return address pushed onto the stack is the start of the encrypted block immediately following the call-site. Using the return address pushed on the stack, the key and the block size, the function decrypts the encrypted block and overwrites it. Once the block has been decrypted, the call to the *Decipher* function is removed by overwriting it with `NOP` instructions.

The decryption function uses the ability to modify code. Therefore, write protection on the code pages is switched off before the modification and switched back on afterwards.

5.5 Experimental Evaluation

We used our obfuscation tool on several malicious programs in order to evaluate its ability to hide trigger based behavior. Although we had to select from a very limited number of available malicious programs written for Linux, we chose programs that are representative of different classes of malware for which triggers are useful.

We evaluated our system by determining how many manually identified trigger-based malicious behaviors were automatically and completely obfuscated as conditional code sections by our system. In order to evaluate the resistance to brute force attacks on each obfuscated trigger, we defined three levels of strength depending on the type of the data used in the condition. An obfuscation was considered *strong*, *medium*, or *weak* if its condition incorporated strings, integers, or boolean flags, respectively. Table 8 shows the results of our evaluation on various programs. Notice that almost all trigger based malicious code was successfully obfuscated using our

Malware	Malicious triggers	Strong	Medium	Weak	None
Slapper worm (P2P Engine)	28	-	28	-	-
Slapper worm (Backdoor)	1	1	-	-	-
BotNET (An IRC Botnet Server)	52	52	-	-	-
passwd rookit	2	2	-	-	-
login rootkit	3	2	-	-	1
top rootkit	2	-	-	-	2
chsh rootkit	4	2	-	2	-

Table 8: Evaluation of our obfuscation scheme on automatically concealing malicious triggers.

tool. However, for a few specific instances our tool was either able to provide weak obfuscation or not able to provide any obfuscation at all. We consider both of these cases as a failure for our obfuscation to provide any protection. We investigated the reasons behind such cases and describe how the malware program could be modified to take advantage of our obfuscation.

We first tested our tool on the Slapper worm [142]. Although it is a worm, it contains a large set of trigger based behaviors found in bots and backdoor programs. When the worm spreads, it creates a UDP based peer-to-peer network among the infected machines. This entire network of victims can be controlled by sending commands to carry out Distributed Denial of Service attacks. In addition, it installs a backdoor program on the infected machine that provides shell access to the victim machine.

The Slapper worm has two components. The first (`unlock.c`) contains the worm infection vector and the code necessary to maintain the peer-to-peer network and receive commands. We manually identified 28 malicious triggers in the program that performs various malicious actions depending on the received control commands. These triggers were implemented using a large `switch` construct. Our tool was able to completely obfuscate all malicious actions of these triggers. However, the obfuscations had medium-level strength because the conditions were based on integers received in the UDP packets. The second part of the worm is a backdoor program (`update.c`)

that opens a Linux shell when the correct password is provided to it. The program contained only one malicious trigger, which uses the `strcmp` function to check whether the provided password was equal to the hard-coded string "aion1981". Our tool was able to successfully obfuscate the entire code of this trigger (which included a call to `execve`) and removed the password string from the program. In this case, our tool provided strong obfuscation because the condition was based on a string.

We next tested our obfuscation on a generic open source bot program BotNET for Linux, which had minimal command and control support built into it. Since bots typically initiate different malicious activities after receiving commands, we identified the sections in that program that receive commands and considered them as malicious triggers. We manually found 52 triggers in the program, and after our obfuscation was applied, code conditionally executed for all 52 of them were strongly obfuscated. This result represents what we can expect by obfuscating any typical bot program. Usually, all IRC based bot commands send and receive text based commands, making the triggers suitable for strong obfuscation.

We found several rootkit programs [109] for Linux that install user level tools similar to trojan horse programs containing specific logic bombs that provide malicious advantages to attackers, including privileged access to the system. First, we tested the `passwd` rootkit program, which had two manually identifiable malicious triggers inserted into the original. The first trigger enables a user to spawn a privileged shell when a predefined magic string "satori" is inserted as the new password. The second trigger works in a similar manner and activates when the old password is equal to the magic string. Our obfuscation successfully concealed these trigger based code with strong obfuscation.

We next tested on a rootkit version of `login`. The source code `login.c` contained three manually identified malicious triggers. Our obfuscation was able to conceal two with strong obfuscation. Both of these triggers were `strcmp` checks on the user

name or password entered by the user. Our obfuscator was not able to protect the third trigger. The reason was that both of the strongly obfuscated code sections increase the value of an integer variable `elite` that was used by the third trigger placed elsewhere in the program. The third trigger used the operator `!=`, making it unsuitable for our obfuscation.

Our next test was on the `top` rootkit. This program contained two malicious triggers, which hide specific program names from the list that a user can view. Although these triggers are implemented using `strcmp`, the actual names of the processes that are to be hidden are read from files. Our obfuscator therefore could not conceal any of these malicious triggers. A malware author could take a different approach to overcome situations like this. By having a trigger that checks for the file containing the process names to start with a hard-coded value, all the other triggers can be contained in a strongly obfuscated code section.

Finally, we tested on a rootkit that is a modified version of the `chsh` shell. Two triggers were manually identified. The first, which checked for a specific username, was strongly obfuscated by our tool. The second trigger used a boolean flag in and therefore was only weakly obfuscated. It is easy for one to overcome this difficulty because by manually modifying the flag to be a string or an integer, stronger obfuscation can be obtained using our tool.

5.6 Discussion

In this section, we discuss our obfuscation technique to help defenders better understand the possible threats it poses. We discuss how malware authors (attackers) may utilize this technique, and analyze its weaknesses to provide insight into how such a threat can be defeated.

5.6.1 Strengths

We have discussed earlier in Section 5.3.4 how our obfuscation impedes state-of-the-art malware analyzers. If the the variable used in the condition has a larger set of possible reasonable values, the obfuscation is stronger against brute force attacks. Since data type information is hard to determine at the binary level, brute-force attacks may have to search larger sets of values than necessary, providing more advantage to the attackers. Equipped with this knowledge, a malware author may modify his programs to take advantage of the strengths rather than naively applying it to the existing programs.

First, a malware author can modify different parts of a program to introduce more candidate conditions. Rather than passing names of resources to system calls, he can query for resources and compare with the names. In addition, certain conditions that use other relational operators such as $<$, $>$ or $\neq$ that are unsuitable for obfuscation may be replaced by $==$. For example, time based triggers that use ranges of days in a month can be replaced with several equality checks on individual days. As another example, a program that exits if a certain resource is not found by using the != operator, can be modified to continue operation if the resource is found using the == operator.

Second, a malware author can increase the size of the concealed code in the malware programs by incorporating more portions of the code under candidate conditions for obfuscation. Bot authors can have an initial handshaking command that encapsulates the bulk of the rest of its activity.

Third, the malware authors can increase the length of inputs to make brute force attacks harder. In our implementation, we use AES encryption with 256-bit keys. If any variable used to generate the key is less than 256-bits in size, then the effective key-length is reduced. Attackers may also avoid low entropy inputs for a given length because that may reduce the search space for dictionary-based attacks. For example,

a bot command that uses lower case characters only in a buffer of 10 bytes needs the search space of size 26^{10}, whereas using both upper, lower, and numeric values would increase search space to 62^{10}.

Unlike strings, numeric values usually have a fixed size and may not be increased in length. Checking the hashes of all possible 32-bit integer values may not be a formidable task, but it is time-consuming, especially if a different nonce is used in each condition to make pre-computed hashes not useful. An attacker can utilize some proprietary function $F(x)$ in the code to map a 32-bit integer x to a longer integer value. However, such an approach does not fundamentally increase the search space. It may just increase the difficulty for the defenders in automatically computing the hashes because the equivalent computation of F has to be extracted from the code and applied for each possible 32-bit value before applying the hash function.

5.6.2 Weaknesses

In its current form, one of the main weaknesses of our obfuscation is the limited types of conditions on which it can be applied to. Although triggers found in the programs that we experimented with were mostly equality tests, there can be many trigger conditions in a malware that checks ranges of values. If the range is large, it may not be possible to represent them as several equality checks as we have mentioned earlier, making the obfuscation inapplicable.

The encryption strength depends on the variable that is used. The full strength of having 2^{256} possible keys for AES is not utilized particularly in the case of numeric data, which are 32-bit or 64-bit integers in current systems. Obfuscations involving string inputs are likely to be more resistant to analysis. Therefore, a subclass of malware, especially bots or backdoors, are likely to be the most beneficial from this approach because the inputs required for the malicious triggers can be selected by the malware authors.

Another weakness is that trigger-based behavior may not just depend on data that are input using system calls, but also status results returned from the calls. Most system calls have a small set of possible return values, usually indicating a success or some form of error. As a result, the number of values to check by a brute force attack may be reduced even further for such conditions.

Possible ways to defeat: If the proposed obfuscation is successfully applied, existing malware analysis techniques may not be able to extract the behavior that is concealed unless the conditions in the triggers are satisfied. Yet, we suggest several techniques to defeat our obfuscation.

First, analyzers may be equipped with decryptors that reduce the search space of keys by taking the input domain into account. Once an obfuscated condition is detected, the variable used in it may be traced back to its source using existing methods. If it is the result or an argument receiving data from a system call, the corresponding specification may be used to find the reasonable set of values that it may take. For example, if the source is set by a calling system call that returns the current system date (such as `gettimeofday`), the set of all possible values representing valid dates may be used, significantly reducing the search space. If, however, the source is input data that cannot be characterized, brute forcing may become infeasible.

Another approach can be to move more towards input-aware analysis. Rather than capturing binaries only, collection mechanisms should capture interaction of the binary with its environment if possible. In case of bots, having related network traces can provide information about the inputs required to break the obfuscation. Existing honeypots already have the capability to capture network activity. Recording system interaction can provide more information about the inputs required by the binary.

Malware detection: Although the obfuscation may prove powerful against malware analyzers, its use may have an upside in malware detection. The existence of

hash functions and encryption routines together with a noticeable number of conditions utilizing them may indicate that an unknown binary is a malware. However, our proposed obfuscation allows more layers of other general obfuscation schemes to be applied on top of it. For example, the binary resulting from our system may be packed with executable protectors [132], which a large fraction of malware already do today. The use of protector tools alone are not usually an indication that the program is a malware because these tools are usually created for protecting legitimate programs. Removing such obfuscation layers require unpacking techniques [122, 76] that are mostly used prior to analysis of suspicious code because of their run-time cost. The end result is that detecting such obfuscated malware is not any easier than detecting existing ones.

5.7 Summary

We have designed an obfuscation scheme that can be automatically applied on malware programs in order to conceal trigger based malicious behavior from state-of-the-art malware analyzers. We have shown that if a malware author uses our approach, various existing malware analysis approaches can be defeated. Furthermore, if properly used, this obfuscation can provide strong concealment of malicious activity from any possible analysis approach that is oblivious of inputs. We have implemented a Linux based compiler level tool that takes a source program and automatically produces an obfuscated binary. Using this tool we have experimentally shown that our obfuscation scheme is capable of concealing a large fraction of malicious triggers that are found in several unmodified malware source programs representing various classes of malware. Finally, we have provided insight into the strengths and weaknesses of our obfuscation technique and possible ways to defeat it.

CHAPTER VI

REVERSING EMULATION-BASED OBFUSCATION

6.1 Motivation

Malware authors often attempt to defeat state-of-the-art malware analysis with obfuscation techniques that are becoming increasingly sophisticated. Anti-analysis techniques have moved from simple code encryption, polymorphism, and metamorphism to multilayered encryption and page-by-page unpacking. One new alarming trend is the incorporation of *emulation technology* as a means to obfuscate malware [150, 107]. With emulation techniques maturing, we believe that the widespread use of emulation for malware obfuscation is imminent.

Emulation is the general approach of running a program written for one underlying hardware interface on another. An obfuscator that utilizes emulation would convert a binary program for a real instruction set architecture (ISA), such as x86, to a bytecode program written for a randomly generated virtual ISA and paired with an emulator that emulates this ISA on the real machine. Figure 14 shows an example of this obfuscation process. Code protection tools such as Code Virtualizer [107] and VMProtect[150] are real-world examples of this class of obfuscators.

Without knowledge of the source bytecode language, many existing malware analysis schemes are crippled in the face of malware obfuscated with emulation. At one end of the spectrum, emulators completely defeat any pure static analysis, including symbolic execution. The code analyzed by static analyzers is that of the emulator; the true malware logic is encoded as bytecode contained in some memory buffer that is treated as data by the analysis. At the other extreme, pure dynamic analysis based approaches that treat the emulated malware as a black box and simply observe

external events are not affected. However, pure dynamic analysis schemes cannot perform fine-grained instruction level analysis and can discover only a single execution path of the malware. More advanced analysis techniques employing dynamic tainting, information flow analysis, or other instruction level analysis fall in the middle of the spectrum. In the context of malware emulators, these techniques analyze the instructions and behaviors of the generic emulator and not of the target malware. As an example, multi-path exploration [101] may explore all possible execution paths of the emulator. Unfortunately, these paths include all possible bytecode instruction semantics and all possible bytecode programs, rather than the paths encoded in the specific bytecode program of the malware instance. In short, we need new techniques to analyze emulated malware.

The key challenges in analyzing a malware emulator are the syntactic and semantic gaps between the observable (x86) instruction trace of the emulator and the non-observable (interpreted) bytecode trace. Theoretically, precisely and completely identifying an emulator's bytecode language is undecidable. Practically, the manner in which an emulator fetches, decodes, and executes a bytecode instruction may enable us to extract useful information about a bytecode program. By analyzing a malware emulator's (x86) trace, we can identify portions of the malware's bytecode program along with syntactic and semantic information of the bytecode language.

In this chapter, we present how automatic reverse engineering of unknown malware emulators is possible. Our goal is to extract the bytecode malware trace (program) and the syntax and semantics of the bytecode instructions to enable further malware analysis, such as multi-path exploration, across the bytecode program. We have developed an approach based on dynamic analysis. We execute the malware emulator and record the entire x86 instruction trace generated by the malware. Applying dynamic data-flow and taint analysis techniques to these traces, we identify data regions containing the bytecode program and extract information about the bytecode

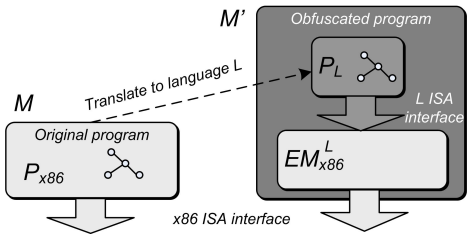


Figure 14: Using emulation for obfuscation

instruction set. Our approach identifies the fundamental characteristic of *decode-dispatch emulation*: an iterative main loop that fetches bytecode based upon the current value of a virtual program counter, decodes opcodes from within the bytecode, and dispatches to bytecode handlers based upon opcodes. This analysis yields the data region containing the bytecode, syntactic information showing how bytecodes are parsed into opcodes and operands, and semantic information about control transfer instructions.

We have implemented a prototype called *Rotalumé* that uses a QEMU [26] based component to perform dynamic analysis. The analysis generates both an instruction trace in an intermediate representation (IR) and a dynamic control-flow graph (CFG) for offline analysis. Rotalumé reverse engineers the emulator using a collection of techniques: abstract variable binding analyzes memory access patterns; clustering finds associated memory reads, such as those fetching bytecode during the emulator's main loop; and dynamic tainting identifies the primary decode, dispatch, and execute operations of the emulator. The output of our system is the extracted syntax and semantics of the source bytecode language suitable for subsequent analysis using traditional malware analyses. We have evaluated Rotalumé on legitimate programs and on malware emulated by VMProtect and Code Virtualizer. Our results show that Rotalumé accurately identified the bytecode buffers in the emulated malware and reconstructed syntactic and semantic information for the bytecode instructions.

6.2 Previous Work

6.2.1 Malware Obfuscation

Malware authors have developed obfuscation schemes designed to impede static analysis [116, 34, 35, 139]. Dynamic analysis approaches that treat malware as a black box can overcome these obfuscation schemes, but they are able to observe only a small number of execution paths. Several approaches have been proposed to address this limitation. Moser et al. proposed a scheme [101] that explored multiple paths during malware execution. Another approach [162] forces program execution along different paths but disregards consistent memory updates. In Rotalumé, these solutions are unable to properly analyze emulated malware because they will explore execution paths of the emulator rather than that of the bytecode program.

Malware authors have broadly applied packing to impede and evade malware detection and analysis. Several approaches based on the general unpacking idea have been proposed [96, 76, 122]. For example, Polyunpack performs universal unpacking based on a combination of static and dynamic binary analysis. Given a packed executable, Polyunpack first constructs a static view of the code. If the executable tries to execute any code that is not present in the static view, Polyunpack detects this as unpacked code.

Recently we observed a new trend in using virtualizers or emulators such as Themida [108], Code Virtualizer [107], and VMProtect [150] to obfuscate malware. These emulators all use a basic interpretation model [137] and transform the x86 program instructions into its own bytecode in order to hide the syntax and semantic of the original code and thwart program analysis. Moreover, by using a randomized instruction set for the bytecode language together with a polymorphic emulator, the reverse engineering effort will have to be applied to every new malware instance, making it very difficult to reuse the reverse engineered information of one emulator for another. We argue that this trend will continue and that a large portion of malware

in the near future will be emulation based. There is no existing technique that can reliably counter an emulation-based obfuscation technique.

6.2.2 Reverse Engineering Known Languages

There are research approaches for analysis and reverse engineering of bytecode for high-level languages such as Java [44, 138]. However, these approaches assume that the syntax and semantics of the bytecode are public or already known. This assumption fails to hold for malware constructed using emulators such as Themida, Code Virtualizer, or VMProtect [108, 107, 150]. These emulators perform a random translation from bytecode to destination ISA, so the connection between the bytecode and final ISA is unknown.

6.2.3 Reverse Engineering Inputs and Protocols

In order to overcome these emulation-based obfuscation techniques, we need analyzers that are able to reverse engineer the emulator model and extract the bytecode syntax and semantics. This is a new research area. In a related area, protocol reverse engineering techniques [164, 90, 37] have been proposed to understand network protocol formats by automatically extracting the syntax of the protocol messages. Tupni [46] automatically reverse engineers the formats of all general inputs to a program. The analysis techniques for extracting the input or network message syntax assume that they can be found at predefined locations in the program. In contrast, one of the main challenges in malware emulator analysis is to find where the bytecode program resides.

6.3 Background

The term emulation generally expresses the support of a binary instruction set architecture (ISA) that is different from that provided by the computer system's hardware. This section describes the use of emulation in program obfuscation and the various

possible emulation techniques.

6.3.1 Using Emulation for Obfuscation

Code authors, including malware authors, are now using emulation to obfuscate programs. In Figure 14, malicious software M consists of a program P_{x86} written in native x86 code and directly executable on x86 processors. Analyzers with knowledge of x86 can therefore perform various analyses on the malware. In order to impede analysis, an adversary can choose a new ISA L and translate P_{x86} to P_L that uses only instructions of L. In order to execute P_L on the real x86 machine, the adversary introduces an emulator EM^L_{x86} that emulates the ISA L on x86. The adversary can now spread a new malware instance M' that is a combination of P_L and EM^L_{x86}.

To further impede possible analysis, a malware author can choose a new, randomly-generated bytecode language for every instance of the malware and make tools to automatically generate a corresponding emulator. Therefore, results found about the bytecode language after reverse engineering one instance of the emulator will not be useful for another instance. Thus, automated reverse engineering of EM^L_{x86} is essential to malware analysis given that each malware instance has a completely unknown bytecode instruction set L and a previously unseen emulator instance.

6.3.2 Emulation Techniques

Various emulation techniques are widely used in software-based virtual machines, script interpretation, run-time abstract interfaces for high-level languages (e.g. Java virtual machine (JVM)), and other environments. Although these are very complex systems, they are all variations of the simple-interpreter method, also known as the *decode-dispatch* approach [137]. Decode-dispatch is used in environments where performance overhead is not an issue. The simple interpreter utilizes a main loop that iterates through three phases for every bytecode instruction: *opcode decode*, *dispatch*,

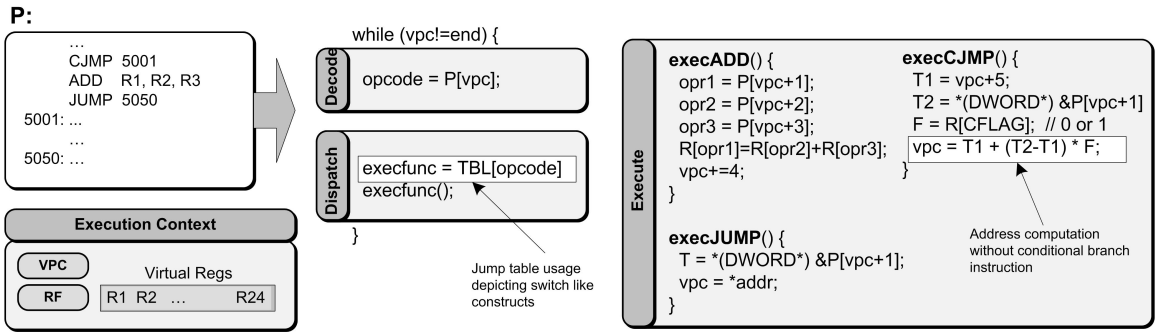


Figure 15: An example of a simple-interpreter (decode-dispatch) based emulator

and *execute*. The decode phase fetches the part of an instruction (opcode) that represents the instruction type. The dispatch phase uses this information to invoke appropriate handling routines. The execute phase, which is performed by the dispatched handling routine, performs additional fetches of operands and executes the semantics of the instruction. We first provide an illustration of how a decode-dispatch emulator works and then discuss the other broad variations in emulation methods.

We show the design of the simple interpreter based emulators using an illustrative running example. Figure 15 shows a fraction of a simple decode-dispatch based emulator [50] written in a pseudo-C like language; it executes programs written in a bytecode language for a hypothetical machine. For conciseness, we describe only the aspects of this machine relevant to the example. This machine supports variable length instructions similar to x86. There are general purpose registers named `R1` to `R24`. A special register called `RF` maintains a flag that can be either 0 or 1 based on some previously executed instruction, and it is used for performing conditional jumps. We show three instructions supported by the machine: `ADD`, `JUMP` and `CJMP`. While the `ADD` instruction takes three operands, both jump instructions take an immediate target address. The conditional jump instruction `CJMP` jumps to the target if `RF` is 1, otherwise control flows to the next instruction.

In this example, the emulator fetches instructions from the emulated program stored in the buffer `P`. An emulator maintains a run-time context of the emulated

program, which includes the necessary storage for virtual registers and scratch space. The emulator maintains execution context via a pointer to the next bytecode instruction to be executed, which we denote throughout the paper as the *virtual program counter* or *VPC*. For the example emulator, the VPC is an index into the buffer `P`. Here, *decoding* is performed by fetching the opcode from `P[VPC]`, i.e. the first byte of the instruction. *Dispatch* uses a jump table resulting from `switch-case` constructs in C. Three *execution* routines for the three instructions are shown in the example. The `execADD` routine updates the register store by adding relevant virtual register values. The `execJUMP` routine updates the VPC with an immediate address contained in the instruction. Finally, `execCJMP` shows how the conditional branch updates the VPC depending on the flag `RF`. It is interesting to note that the branch is emulated without using any conditional jump, but rather with a direct address computation. This shows how an emulator provides a way to remove identifiable conditional branches, making it hard for analysis approaches such as multi-path exploration to even explore any branch related to the emulated program.

More sophisticated emulation approaches often improve efficiency. The *threaded approach* [80] improves performance by removing the central decode-dispatch loop. The decoding and dispatching logic is appended to the execution semantics by adding a copy of that code to the end of each execution routine. This removes several branches and improves execution performance on processors that have branch prediction. By using *pre-decoding* [93], the logic of decoding instructions to their opcodes and operands executes only once per unique instruction, and the program subsequently reuses the decoded results. Hence, the opcode decoding phase is not executed for each executed bytecode instruction. The *direct threading approach* [25] removes jump table lookups by storing the function address that executes the instruction semantics together with the predecoded results. Therefore, the dispatch of the next

instruction's execution routine is an indirect control transfer at the end of the previous bytecode instruction's routine. Finally, *dynamic translation* [135], one of the most efficient methods of emulation, converts blocks of the emulated program into executable instructions for the target machine and caches them for subsequent execution. These categories of emulators maintain a VPC that is updated after blocks of the translated native instructions are executed.

Dynamic translation based emulators have very complex behavioral phases. They may seem attractive to malware authors because of their analysis and reverse engineering difficulty. However, like page-level unpacking used in some packers [132], dynamic translation reveals large blocks of translated code as a program's execution proceeds. This approach reduces the advantage of using emulation as an obfuscation because the heuristics used by automated unpackers can capture the fact that new code was generated and executed. In this paper, we focus our methods on automatically reverse engineering the decode-dispatch class of emulators.

Several challenges complicate automatic reverse engineering of interpreter-based emulators. First, no information of the bytecode program, such as its location, are known beforehand. Second, no information regarding the emulator's code corresponding to the decode, dispatch, and execution routines is known. Finally, we anticipate that emulator code varies in terms of how it fetches opcodes and operands, maintains context related to the emulated program, dispatches code, executes semantics, and so on. An adversary may even intentionally attempt to complicate the identification of bytecode by storing the bytecode program in non-contiguous memory or use multiple correlated variables to maintain the VPC.

Our current work, as a first step, advances the state-of-the-art and significantly challenges attackers. It also lays the foundation for reverse engineering of emulators that are based on other (more advanced and efficient) approaches.

6.4 Reverse Engineering of Emulation

In order to enable malware analysis of an emulated malware instance, it is necessary to understand the unknown bytecode language used by the instance. We have developed algorithms to systematically and automatically extract the syntax and semantics of unknown bytecode based upon the execution behavior of the decode-dispatch based emulator within a malware instance. Our approach identifies fundamental characteristics of decode-dispatch emulation: an iterative main loop, bytecode fetches based upon the current value of a virtual program counter, and dispatch to bytecode handlers based upon opcodes within the bytecode. This analysis yields the data region within the malware containing the bytecode, syntactic information showing how bytecode instructions are parsed into opcodes and operands, and semantic information consisting of native instructions that carry out the actions of the bytecode instructions. We identify the control-flow semantics from which structures such as a control-flow graph (CFG) can be generated. Together, these techniques provide the foundation for overcoming the emulation layer and performing subsequent malware analysis.

Our algorithms are based on dynamic analysis. We execute the emulated malware once in a protected environment and record the entire x86 instruction trace generated by the malware. From this trace, we extract syntactic and semantic information about the bytecode that the malware's emulator was interpreting during execution. The contributions made by our approach offer the opportunities to reconstruct behavioral information about unknown bytecode interpreted by an unknown decode-dispatch emulator and to subsequently apply traditional malware analysis to the sample.

The algorithms operate as follows:

1. Identify variables within the raw memory of the emulator based upon the access patterns of reads and writes in an execution trace. We developed *abstract variable binding*, a forward and backward dynamic data-flow analysis, for this

identification.

2. Identify the subset of those variables that are candidates for the emulator's virtual program counter (VPC). We find possibilities by clustering the emulator's memory reads, some of which are bytecode fetches, based upon the abstract variable used to specify the accessed memory location.

3. Identify the boundaries of the bytecode data within the x86 application, the decode-dispatch loop, and the emulator phases. For each cluster of reads through the same abstract variable, we determine whether the reads occurred during execution of loop iteration with emulator-like operations.

4. Identify the syntax and the semantics of bytecode operations. We examine how bytecode is accessed by the emulation phases to identify its syntax. We discover bytecode handler, forming the semantics of the bytecode instructions. Handlers that cause non-sequential updates to the VPC indicate that the bytecode opcode corresponds to a control transfer operation, allowing our system to construct a CFG for the bytecode.

The following sections describe these steps in detail.

6.4.1 Abstract Variable Binding

A decode-dispatch emulator fetches bytecode instructions from addresses specified by a virtual program counter (VPC). Like a program counter or instruction pointer register in hardware, a VPC acts as a pointer to the currently executing bytecode instruction. Knowing the memory location of the VPC allows an analyzer to observe how the emulator accesses bytecode instructions and executes them, which reveals information about the bytecode instruction syntax and semantics. We locate an emulator's VPC through a series of analyses, beginning with abstract variable binding.

Abstract variable binding identifies, for each memory read instruction of an execution trace, the program variable containing the address specifying the location from which the data should be read. Consider pseudo-code of an emulator that regularly fetches instructions pointed to by the VPC:

$$\texttt{instruction} = \texttt{bytecode}[\texttt{VPC}]$$

or

$$\texttt{instruction} = *\texttt{VPC} \tag{1}$$

In these examples, the VPC is an index into an array of bytecode or a direct pointer into a buffer of bytecode. During its execution, the emulator will execute these bytecode fetches many times. Although each fetch may access a different memory location within the bytecode buffer, all fetches used the same VPC variable as the specifier of the location. Abstract variable binding will attach a program variable, such as VPC, to every memory read instruction in the execution trace that uses that variable to specify its access location.

Successful abstract variable binding will help our analyzer identify the VPC and the bytecode buffer used by the unknown emulator in a malware instance. Each bytecode fetch will appear in the execution trace as a memory read instruction whose accessed location is bound to the VPC variable. The emulator likely executes many other memory reads unrelated to bytecode fetch, and these may have their own bindings to other variables in the program. Steps 2 and 3 of our algorithms, presented in Sections 6.4.2 and 6.4.3, whittle down the bindings to only those of the VPC.

Our analysis of x86 instruction traces rather than source code complicates abstract variable binding in fundamental ways. First, a binary program has no notion of high-level language variables. A compiler translating an emulator's high-level code into low-level x86 instructions will assign each variable to a memory location or register in a way unknown to our analysis. Second, the x86 architecture requires all memory

read and write operations accessing dynamically computed addresses to use register indirect addressing. In case of performing a memory read using an address stored in a variable, if the variable is assigned a memory location, then that value will be transferred into the register rather than being accessed directly. For example, the pseudo-x86 translation of (1) may be the two-instruction sequence:

$$\texttt{eax} \leftarrow [\texttt{VPC}] \tag{2}$$

$$\texttt{instruction} \leftarrow [\texttt{eax}] \tag{3}$$

where `VPC` represents a pointer variable (assigned a particular memory address), `eax` is a register, and `instruction` is a register or memory location. The first instruction loads the value of the variable `VPC` to `eax`. This value is the address used in the second instruction where the register `eax` can be considered temporary storage for the `VPC`. In other words, (2) binds `eax` to `VPC`, and this binding is propagated to the read operation in (3).

With a limited number of registers, the same register can be bound to different variables at different points of execution. When updating a variable with a non-immediate value, the new value must be loaded into a register using an instruction similar to (2) before transferring the value to the variable's assigned memory location. In this case, the register is already bound to a variable and the binding is not changed when loading the value into the register. Without knowledge of how variables in the program are assigned to memory or registers, it is hard to determine whether a register load operation such as (2) is a new variable binding to the register or a new value assigned to an already bound variable. We draw three conclusions that impact the design of our abstract variable binding algorithm. First, we use absolute memory addresses as our description of a high-level language variable. Second, we must analyze data flows along an entire trace to determine variable binding information appropriately. Third, to be able to identify all abstract variables, we must

conservatively consider both scenarios described above for each instruction similar to (2).

Abstract variable binding propagates binding information using dynamic data-flow analysis across an execution trace of the emulated malware. We use both a forward and a backward propagation steps, corresponding to the two different ways in which a variable's value may be set. A variable's value may be an incremental update to its previous value; forward binding identifies the abstract variables (memory locations) from which a read operation's address is derived in this case. A variable's value may also be directly overwritten with a new value unrelated to its previous contents. Backward binding determines the appropriate bindings in this case based on the variable's future use. Our backward binding algorithm is conservative and introduces imprecision.

A malicious emulator may also attempt to complicate analysis by obfuscating its use of a VPC. For example, it may replace a single VPC with a collection of variables and switch among the collection during execution. However, the collection must together still follow an orderly progression and update sequence to ensure that the emulator correctly executes the bytecode. This fundamental need to maintain consistency will produce data flows between two elements of the VPC collection whenever the emulator switches from one element to another. Our data-flow analysis will track these flows and remains robust to this type of emulator obfuscation.

We use the following notations for the presentation of our algorithms. Let M denote the memory address space of the emulator and R the set of registers available on the target hardware. We uniquely identify abstract variables by memory addresses using the set $\alpha \subseteq M$. We represent instructions in an intermediate representation that expresses only simple move and compute operations on registers and memory and has one source and one destination. Let the loading of constant c into register r be denoted as $r \leftarrow c$. A memory read operation is $r \leftarrow^{v} [m]$, which indicates that the

Algorithm 1 Forward Binding

Initialize: $\forall r \in R : B_0(r) = \{\}$ and $V_0(r) = \texttt{NONE}$
for $i = 1$ to l **do**
  $\forall r \in R : B_i(r) = B_{i-1}(r)$ and $V_i(r) = V_{i-1}(r)$
  Update B_i and V_i using rules F1 to F6
  if $i \in \rho$ and instruction is $r_1 \leftarrow^v [r_2]$ or $r_1 \twoheadleftarrow^v [r_2]$ **then**
    $FB(i) = B_i(r_2)$
  end if
end for

value v is read from memory address m and loaded into register r. Here m can be a constant or register holding an address. Memory writes are denoted by $[m] \leftarrow^v r$. A register-to-register move operation is $r_1 \leftarrow r_2$. If the right-hand side of any of these operations involves computation using the specified variable or address, then we denote the assignment as $\twoheadleftarrow$ rather than $\leftarrow$. We identify each instruction in an execution trace with a sequence number $i \in \mathbb{N}$. Let the set $\rho \subseteq \mathbb{N}$ consist of all read operations of the form $r \leftarrow [m]$ or $r \twoheadleftarrow [m]$. Lastly, let the function $Addr : \rho \rightarrow M$ be defined as: $Addr(i)$ represents the address of a read operation $i \in \rho$.

6.4.1.1 Forward Binding

Forward binding identifies those variables supplying the address used by a read operation. The algorithm works on the execution trace and propagates information forward along the instructions in the trace. For each register, we track both the set of abstract variables bound to the register and the value stored in the register. A memory read instruction is bound to the same variable as that bound to the register specifying the address of the read. The set $B_i(r)$ denotes the abstract variables bound to register $r \in R$ at instruction i. $V_i(r)$ represents the value in register r at i. Forward propagation updates bindings and values according to the following six rules based upon the instruction type. For convenience, we use the function f_i to indicate the computation of instruction i.

F1 $r \leftarrow c$: $B_i(r) = \{\}$ and $V_i(r) = c$

Algorithm 2 Backward Binding

Input: $\forall r \in R$ and $1 \leq i \leq l : V_i(r)$ from Alg. 1
Initialize: $\forall r \in R : B'_{l+1}(r) = \{\sigma_r\}$ for $1 \leq j \leq k$
for $i = l$ to 1 **do**
  $\forall r \in R : B'_i(r) = B'_{i+1}(r)$
  Update B'_i using rules B1 to B3
  if $i \in \rho$ and instruction is $r_1 \leftarrow^v [r_2]$ or $r_1 \twoheadleftarrow^v [r_2]$ **then**
    $BB(i) = B'_i(r_2)$
  end if
end for

F2 $r \leftarrow^v [c]$: $B_i(r) = \{c\}$ and $V_i(r) = v$

F3 $r_1 \leftarrow^v [r_2]$: $B_i(r_1) = \{V_{i-1}(r_2)\}$ and $V_i(r_1) = v$

F4 $r \twoheadleftarrow c$: $B_i(r) = B_{i-1}(r)$ and $V_i(r) = f_i(V_i(r), c)$

F5 $r \twoheadleftarrow^v [c]$: $B_i(r) = B_{i-1}(r) \cup \{c\}$ and $V_i(r) = f_i(V_{i-1}(r), v)$

F6 $r_1 \twoheadleftarrow^v [r_2]$: $B_i(r_1) = B_{i-1}(r_1) \cup \{V_{i-1}(r_2)\}$ and $V_i(r_1) = f_i(V_{i-1}(r_1), v)$

Rules F1–F3 apply new register value assignments where values are taken directly from a constant, register, or memory location. Bindings reset to those of the data source. Rules F4–F6 compute assigned values from the previous register value and other data; these correspond to incremental updates to a value and do not cause bindings to reset. Rules F3 and F6 correspond to reads using indirect addressing and are points where instruction bindings indicate possible VPC use.

All rules update B and V based on the current and immediate predecessor instructions, so propagation operates from the first instruction to the last. The algorithm outputs the mapping $FB : \mathbb{N} \rightarrow \alpha^*$ describing bindings from memory read operations to abstract variables. Algorithm 1 presents the pseudo-code for forward binding; l is the trace length.

6.4.1.2 Backward Binding

We expect realistic bytecode languages to provide one or more types of control transfer operations such as jumps and branches. Forward binding propagates abstract variable bindings when operations compute an incremental update to an already-bound variable, which does not reflect the semantics of a control transfer. Control transfers will instead cause the emulator to directly overwrite the VPC with the target address. The emulator will execute instructions matching rule F1 or F2, which reset the binding information. Backward binding, a second variable binding step that follows the forward algorithm, computes bindings in such cases by identifying bound variables when a value is *written out* to memory and then propagating the binding backward to all previous uses. This algorithm is conservative and may over-approximate the actual variable bindings.

Backward binding operates in the reverse order of forward binding. It first binds a variable to a register when that register's value is stored to memory, and then propagates the binding backwards to other registers using the following rules:

B1 $[r_1] \leftarrow r_2 : B'_i(r_2) = \{V_i(r_1)\}$

B2 $[c] \leftarrow r : B'_i(r) = \{c\}$

B3 $r_1 \leftarrow r_2 : B'_i(r_2) = B'_{i+1}(r_1)$

The algorithm outputs the mapping for backward binding $BB : \mathbb{N} \rightarrow \alpha^*$. Algorithm 2 presents pseudo-code for backward binding.

6.4.1.3 Identifying Dependent Abstract Variables

Variables used to contain memory read addresses may themselves be interdependent. For example, if the obfuscation technique of utilizing a collection of VPCs is used, the VPC related variables have a relationship with each other. Likewise memory reads that access operands may occur at short, fixed offsets from the VPC value. We

identify dependencies among abstract variables by tracking the flow of data values from one abstract variable to another. As this is forward data-flow, we incorporate dependent variable identification as part of our forward binding algorithm.

Let the mapping $DV : \alpha \rightarrow \alpha^*$ denote abstract variable dependencies, where $Y \in \alpha$ is a dependent variable of X if $Y \in DV(X)$. To populate this mapping, we add the following rule to Algorithm 1:

D1 $[r_1] \leftarrow r_2 : DV(r_1) = DV(r_1) \cup \{B_i(r_2)\}$

Dependencies are commutative. After the completion of Algorithm 1, we add the converse of each dependency identified by the algorithm: $\forall X, Y \in M$: if $X \in DV(Y)$ then add $Y \in DV(X)$. Identifying these abstract variable dependencies thwart attacks that introduce extraneous store operations or copy operations from one variable to another before use.

6.4.1.4 Lifetime of Variables

Our x86 execution trace analysis introduces one final challenge that is often not present when using high-level language variables. We use absolute memory locations as our abstract variables, but the same memory location may be used for different variables at different points of execution. Although variables in the static data region have the lifetime of the entire execution, variables on the stack and heap have shorter lifetimes. The same address can be shared among multiple variables in different execution contexts depending on the allocation and deallocation operations performed during execution.

We address the limited lifetime of stack variables by including stack semantics and analysis of the stack pointer register `esp` as part of our algorithms. Our set of abstract variables α is made of tuples $\alpha \subseteq M \times \mathbb{N} \times \mathbb{N}$ that use integers to denote the start and end of the variable's live range within the execution trace. At the first access to a memory address $m \in M$ at instruction s, we add (m, s, ∞) to α. If the t^{th}

instruction modifies the `esp` register such that an abstract variable's memory address has been deallocated from the stack, then the end of its lifetime is set to t. Any access to the same memory address after its lifetime expired creates a new abstract variable. For pedagogy, we presented Algorithms 1 and 2 without live ranges; updates to the algorithms to include lifetimes are straightforward.

In this work, we do not address lifetimes for variables allocated on the heap. Prior heap analysis research [22] often assumed that the analyzer understood the heap allocation and deallocation routines. We cannot make this assumption for malware binaries, which may be stripped of debugging information and deliberately obfuscate the heap routines. Further research is needed to address this open problem.

6.4.2 Identifying Candidate VPCs

We use the computed variable bindings to identify candidate variables that may be the malware emulator's virtual program counter. We first combine the bindings identified by the forward and backward algorithms to compute the complete abstract variable bindings for each memory read. Let the function $\mathit{Vars} : \mathbb{N} \rightarrow \alpha^*$ be computed as the transitive application of the dependence function DV to $FB(i) \cup BB(i)$ for a read operation $i \in \rho$. We then cluster all read operations within the execution trace and group together those reads that are bound to common abstract variables. Our clustering uses a simple similarity metric that treats two reads $i_1, i_2 \in \rho$ as similar if $\mathit{Vars}(i_1) \cap \mathit{Vars}(i_2) \neq \emptyset$, and dissimilar otherwise. The clustering algorithm will output n clusters $C_1, \ldots, C_n$ where each cluster C_i is a set of read operations.

The malware bytecode should be fetched for execution exclusively by memory read operations contained within one of the n clusters. Abstract variable binding over-approximates actual bindings due to the backward algorithm, which results in two reads clustered together if they *may* use the same abstract variable to specify the accessed address. The transitive closure of the dependencies among abstract variables

ensures two reads will be similar even if the reads use two distinct abstract variables. Therefore, the bytecode program will be completely contained within a cluster. Each cluster is then a candidate collection of instruction fetches into bytecode, and the common abstract variables at each cluster are candidate VPCs.

6.4.3 Identifying Emulation Behavior

We analyze each candidate cluster and VPC to find a cluster containing memory reads characteristic of emulation. Decode-dispatch emulators have fundamental execution properties: a main loop with a bytecode fetch through the VPC, decoding of the opcode within the bytecode, dispatch to an opcode handler, and a change to the VPC value. For each candidate cluster, we hypothesize that the memory region read by the cluster corresponds to bytecode and then test that hypothesis. We determine whether there exists an iterative pattern of bytecode fetches through the associated candidate VPC and updates to that possible VPC. To detect loops, we first create a partial dynamic control-flow graph (CFG) of the program in execution. We use the control-flow semantics of the executed instructions to create new basic blocks and split already created blocks. We use function call semantics to create separate CFGs for each function. Then, we use the standard loop detection methods used for static intra-procedural CFGs [15].

To find decoding, dispatching, and execution of bytecode after the memory read fetches it from the bytecode buffer, we analyze how read values are used by other instructions within the execution trace. We use multi-level dynamic tainting [168] to track the propagation of the data read from instructions in a candidate cluster through the emulator's code. In contrast to traditional taint analysis with 0/1 taint labels, we apply multiple labels to memory contents and registers at the byte level. Different labels track individual data read from the cluster and maintain state information related to which phase—fetch, decode, dispatch, or execute—that the emulator may

be in for a particular read.

We use dynamic taint analysis as follows. For each candidate cluster, we taint the data bytes in the hypothesized bytecode buffer region of the cluster with the label $\langle opcode, id \rangle$ where an id is a unique per-byte identifier. When a read operation in the execution trace accesses a tainted byte, we mark the instruction as an *opcode fetch* for the particular id in the label. If the instruction sequence number is i, then we also taint the forward bound variables $FB(i)$ and the register holding the address accessed by the read operation with the label $\langle vpc, id \rangle$, indicating that it is a VPC for the emulator. Execution continues until our analyzer detects opcode dispatch behavior.

We identify dispatch behavior by looking for control-flow transfer instructions executed by the emulator that are influenced by data read from the cluster's hypothesized bytecode buffer: these are transfers into handlers for specific bytecode opcodes. In the simplest scenario, an x86 instruction like `jmp` or `call` can target an address read from a tainted register or from a dispatch table accessed through a tainted register. More complex code patterns may include arbitrary data and control flows between a control-flow target lookup and the actual dispatch. Taint propagation ensures that taint labels transfer from address to values read through that address, to copies of those values, and to the control-flow transfer. Once the analyzer detects a dispatch-like behavior, it marks the dispatch instruction with the id of the taint label and the analysis now tracks the target of the control-transfer as a probable *execute* routine.

Each subsequent read in the candidate cluster may be accessing a new bytecode, operands for the current bytecode, or an unrelated memory value included in the cluster due to the imprecision of backward variable binding. We first identify new bytecode accesses by analyzing the dynamic CFG to see if execution looped since the previous bytecode fetch. If a loop is not detected, we then check to see if the read is accessing a probable operand in the bytecode buffer. If the register used to perform

the read operation is tainted as $\langle vpc, id \rangle$ with the id of the current iteration, and the computation of the accessed address added a small constant to the candidate VPC value, then the memory access is likely for an operand. We consider all other accesses to be spurious.

We consider every candidate cluster containing iterative memory reads in a loop that includes dispatch behavior. There must be at least two loop executions in the dynamic trace for our analysis to identify the loop.

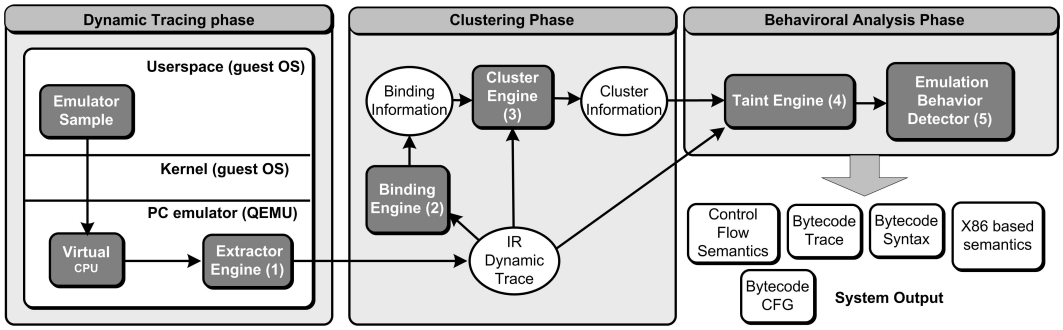


Figure 16: Analysis process overview

6.4.4 Extracting Syntax and Semantics

Once the analyzer identifies the emulation behavior, it reverse engineers each iteration of the emulator loop to extract the syntax and semantics of the bytecode instruction executed on that iteration. The syntax of bytecode details how to parse the instruction: its length and the placement of its opcode and operands. Bytecode semantics describe the bytecode's effect upon execution of the malware instance. We are particularly interested in identifying bytecode instructions exhibiting control-flow transfer semantics, as these are the locations where malware analysis techniques such as multipath exploration [101] should be applied.

We identify the syntax of bytecode instructions by observing the memory reads made from the data regions containing bytecode, as determined in Section 6.4.3. To identify the opcode part of the instruction, we apply our taint analysis to determine

which portion was used by the emulator's dispatch stage for selection of an execution handler. We can identify opcodes at the granularity of one or more bytes within a bytecode instruction, as our taint analysis works at byte-level. An emulator may dispatch several different opcodes to the same execution routine because their semantics may be similar. As a result, we count the number of instructions in the bytecode instruction set as the number of unique execution routines identified in our analysis.

The execution routine invoked by the emulator for the bytecode's opcode encodes the semantics of the opcode. We find control flow transfers by analyzing the changes made by an execution routine upon the VPC of the emulator. Unconditional transfers, including fall-through instructions, will always set the VPC to the same value on every execution of that instruction. Commonly, fall-throughs simply advance the VPC to the next instruction in sequential order, a regular update pattern that can be readily identified. Conditional control-flow transfers and transfers to dynamically-computed targets will update the VPC in different ways upon repeated execution of the bytecode instruction.

By determining how to parse the bytecode buffer and by locating control-flow transfer opcodes, we are then able to construct a control-flow graph (CFG) for the bytecode. The locations of the control-flow transfers and their target addresses within the bytecode stipulate how to divide the entire bytecode buffer into basic blocks. The transfers then produce edges between the blocks corresponding to possible VPC changes during emulated execution. The CFG structure provides a foundation for subsequent malware analysis.

6.5 Implementation

Our automatic reverse engineering occurs in three different phases: *dynamic tracing*, *clustering*, and *behavioral analysis*. Figure 11 shows the different phases of our process and the interactions among the architectural components used by the different

analysis steps. The dynamic tracing phase gathers run-time data related to a malware emulator's execution, and allows the clustering and behavioral analysis phases to extract malware bytecode and the syntactic and semantic information for the bytecode instruction set.

There are two important requirements for the run-time environment of the dynamic tracing phase: instruction-level tracing, and isolation from malware and attacks. Since the analysis techniques in Rotalumé are orthogonal to the underlying run-time environment and our goal here is to develop and evaluate these techniques, we implemented our dynamic analysis techniques on top of QEMU [26], which emulates an x86 computer system. For a deployable version of Rotalumé, we suggest using a more transparent and robust environment, such as a hardware virtualization based system like Ether [51]. The components in the latter two phases were developed as an offline analyzer written in C++. In our current prototype implementation, each individual phase is activated manually using the result of the previous phase. However, our design can be completely automated to process large numbers of malware samples.

6.5.1 Dynamic Tracing

The first phase collects the dynamic instruction trace of the emulator program that is executing as the QEMU guest operating system. We modified QEMU by inserting a callback function that invokes Rotalumé's *Trace Extractor Engine* (EE) for every instruction executed in QEMU. The EE component collects necessary context information related to the executed instruction and stores the intermediate-representation (IR) that is used in latter phases of the system. Our IR is self-contained—we store the instruction representation as well as the values of the operands involved in the instruction. We log all information so that we may perform off-line analysis without requiring additional dynamic analysis. The output information of this phase is

represented by the dynamic trace of the program in IR form.

6.5.2 Clustering

The second phase clusters the memory read operations visible in the trace. We group together every read operation performed by the program based on the common variable used to access that read memory location. This phase is performed by two main components: the *Binding Engine* (BE) and *Clustering Engine* (CE). The BE component is a program that takes as input the IR dynamic program trace and applies the *backward* and *forward* abstract variable binding algorithms described in Section 6.4.1. For each algorithm, we store binding information differently. More specifically, for each instruction in forward binding, we store the following information: *instruction id* (a unique identifier for each instruction present in the dynamic program trace IR), the *destination register operand* of the instruction, and the bound variables associated with the destination register according to the rules described in Section 6.4.1.1. For each instruction in backward binding, we store the *instruction id* and the *bound variables* associated with the registers or memory locations according to the rules defined in Section 6.4.1.2. The BE component provides the binding information to the CE. The CE component is a program that inputs the IR dynamic trace and the binding information, and applies the clustering algorithm. At a high level, CE takes the union of forwarding and backward binding information, applies the dependence function in Section 6.4.2, and provides the cluster information. The cluster information contains a vector of sets where each set contains the addresses of the memory read instructions that are accessed by the same variable. At the end of this phase, the cluster information is saved to a file.

6.5.3 Behavioral Analysis

The behavioral analysis phase provides the final information output of Rotalumé. We implemented a behavioral analyzer composed of two sub-components: the *Taint*

Engine and the *Emulation Behavior Detector*. The behavioral analyzer is a program that takes as input the IR dynamic trace and clustering information, and analyzes one cluster at a time. For each cluster, the Taint Engine taints the memory address contained in the cluster and activates the Emulation Behavior Detector. This analyzer is a state machine that follows the tainted addresses and identifies the emulation behavior, as described in Sections 6.4.3 and 6.4.4. Whenever the analyzer recognizes an opcode, the system stores information of the opcode into a file. More specifically, the analyzer stores for each opcode executed: its opcode value, the operands' values, and the x86 code in assembly format associated with the executed opcode.

6.6 Evaluation

We evaluated Rotalumé using both synthetic and real programs that include both legitimate applications and malware, including real-world emulated malware. These programs are obfuscated to run on three commercially available packers that support emulation: *Code Virtualizer* [107], *Themida* [108], and *VMProtect* [150]. VMProtect and Code Virtualizer convert selective functions of a given binary program into a bytecode program with a randomly generated bytecode language. Themida, which is more widely used for malware, does not apply emulation to the given malicious binary program but rather to the unpacking routine and the code that invokes API calls.

6.6.1 Synthetic Tests

We first experimented with synthetic test programs. Our goal was to use the ground truth of the synthetic programs to evaluate the information about the extracted bytecode program and the syntax and semantics of the virtual instruction set architecture identified by Rotalumé. We used Code Virtualizer and VMProtect because they can obfuscate any user-specified function in a program.

We wrote three simple synthetic test programs in C. Each test program contained

Table 9: Description of synthetic test programs

Program	Description	x86 Program		x86 Trace	
		Inst.	C-Flow	Inst.	C-Flow
`synth1`	No branch	24	1	24	1
`synth2`	Nested `if`	61	11	21	7
`synth3`	Loop and `if`	55	10	270	54

Table 10: Results for synthetic programs obfuscated with Code Virtualizer

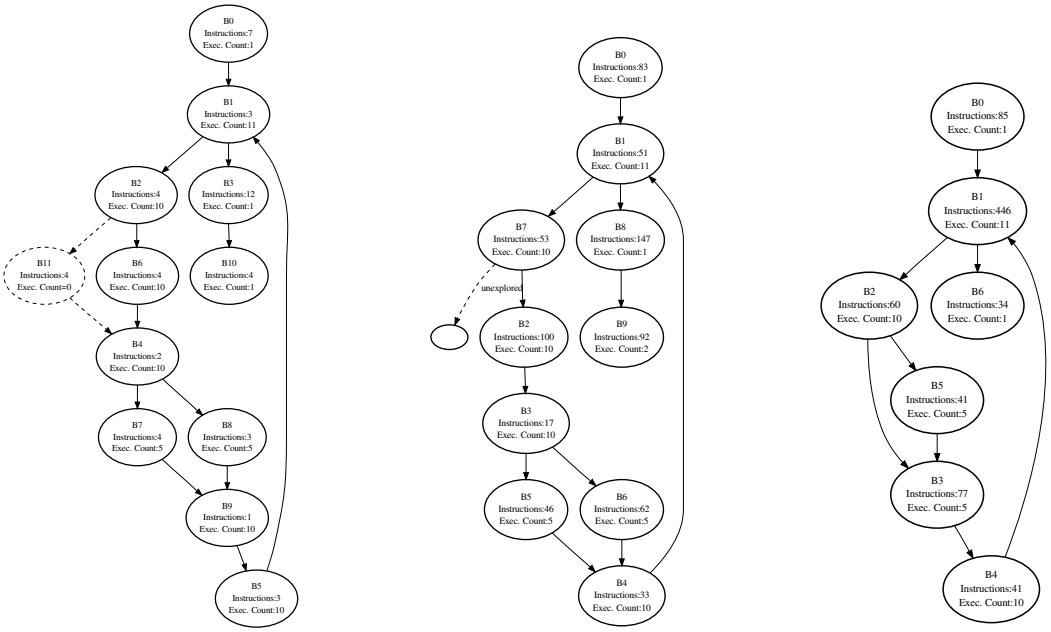
Program	Bytecode Trace (inst. count)		Bytecode Program (inst. count)		Virtual Instruction Set Architecture				
	All types	C-Flow	All types	C-Flow	All types	C-Flow (Cond.)	0 Opr	1 Opr	2 Opr
`synth1`	277	1	277	1	23	1 (0)	3	6	15
`synth2`	254	7	254	7	27	3 (1)	4	8	16
`synth3`	3481	54	684	8	31	3 (1)	4	9	18

Table 11: Results for synthetic programs obfuscated with VMProtect

Program	Bytecode Trace (inst. count)		Bytecode Program (inst. count)		Virtual Instruction Set Architecture				
	All types	C-Flow	All types	C-Flow	All types	C-Flow (Cond.)	0 Opr	1 Opr	2 Opr
`synth1`	497	1	497	1	16	1 (0)	4	8	4
`synth2`	442	7	442	7	18	2 (0)	4	9	5
`synth3`	5709	54	785	8	18	2 (0)	4	9	5

a function with distinguishable control-flow characteristics that we wanted to obfuscate. We compiled these programs and converted them to x86 binaries. We analyzed the static characteristics of the compiled code using IDAPro [5] and the dynamic characteristics by tracing the programs in our QEMU-based system. Table 9 lists information about the three functions of these test programs. For each function, the table shows the total numbers of x86 instructions ("Inst.") and control-flows instructions ("C-Flow") obtained from static analysis. The total numbers of x86 and control-flow instructions in an execution trace of the functions, obtained from dynamic analysis, are also shown. Program `synth1` involves simple computation without any conditional branch or function. Program `synth2` contains nested `if` statements, and hence its execution trace contains only a part of its (static) program instructions. Finally, `synth3` contains both `if` statements and a `for` loop. Its trace length was larger than the static x86 instruction count because of loops.

We used VMProtect and Code Virtualizer to obfuscate the selected functions in our three test (binary) programs. We then applied Rotalumé to analyze them.



(a) Original x86 CFG

(b) Bytecode CFG (Code Virtualizer)

(c) Bytecode CFG (VMProtect)

Figure 17: Comparison of x86 and bytecode CFGs of the `synth3` test program

Rotalumé was able to correctly identify emulation behavior in all of the test cases, and Tables 10 and 11 summarize respectively the results of reversing Code Virtualizer and VMProtect. The results show information for bytecode instructions traced and identified at run-time in terms of the instruction counts (of all types and the control-flow instructions) of the bytecode execution trace and the program itself. The results also show the virtual instruction set architecture (ISA) discovered by Rotalumé in terms of the number of unique bytecode instructions, information regarding the syntax of the bytecode language in terms of number of operands, and information regarding the semantics of conditional control-flow transfers.

In both VMProtect and Code Virtualizer, the bytecode trace of a program was significantly longer than its original x86 binary. For example, `synth3` executed 3,481 bytecode instructions of Code Virtualizer and 5,709 of VMProtect, compared to just 270 x86 instructions in the original program. The results also show that for all test

cases, Rotalumé accounted for the same number of control-flow instructions in the bytecode execution trace as in the original x86 execution trace. This shows that Rotalumé was able to extract the control-flow information of the original programs.

Table 11 shows that for both `synth2` and `synth3`, the VMProtect virtual ISA extracted by Rotalumé does not have conditional control-flow instructions, unlike the results from Code Virtualizer. We investigated this discrepancy by analyzing the x86 execution traces of VMProtected software and then comparing with the bytecode information provided by Rotalumé. We found that Rotalumé correctly identified the decode, dispatch, and execution routines of the emulator. We manually analyzed the traces of the execution routines and did not find any x86 conditional branch instruction. This means that there were no conditional jumps in the bytecode program traces. By carefully analyzing the semantics of the instructions before the control-transfer instruction, we confirmed that VMProtect emulates conditional branches by dynamically computing the target address and using a single jump instruction.

Figure 17 shows that the control-flow graphs extracted by Rotalumé for `synth3` are very similar to that of the original x86 program. Figure 17(a) shows the original x86 program's CFG, and it contains a loop with two conditional branches. The graph shows that basic block `B11` was not executed during execution. Figure 17(b) shows the CFG of the Code Virtualizer bytecode program trace as extracted by Rotalumé. The two CFGs show identical control-flow semantics. Interestingly, we also could identify that there is an unexplored path from basic block `B7`. This was possible because Code Virtualizer's bytecode language has a conditional branch instruction that was identified by Rotalumé even though it was not executed. This shows a key benefit of our approach: other analyses such as multipath exploration [101] can be selectively applied to explore such paths in the emulated malware bytecode rather than in the entire emulator.

The CFG in Figure 17(c) is for the VMProtect bytecode trace. Since we found that

Table 12: TR/Killav.PS obfuscated by VMProtect

Description	Dyn. x86 CFG Inst. (BB)	Dyn. BC CFG Inst. (BB)
Original	1528 (435)	×
1 function packed	3618 (738)	2617 (16)
5 functions packed	4103 (801)	3830 (49)

VMProtect's bytecode has no explicit conditional branches, we are unable to provide information about a possible path that was not executed in the trace. However, we can identify the dynamically computed control-flow instructions in the trace and mark where analysis of possible branch target addresses needs to be applied. Thus, we can still uncover the control-flow information of the bytecode program. The CFG shows the existence of the loop and the condition but the number of basic blocks is fewer than the original CFG. This likely occurs because VMProtect applies optimization on the bytecode.

6.6.2 Real (Unpacked) Programs

We next tested on a real program obfuscated with emulation by comparing the extracted bytecode information against the original x86 program. We selected a malware program that is not packed because self-modifying code can not be translated into bytecode. We randomly selected the *Killav.PS* malware identified as a Trojan by *Avira Antivir* antivirus software [17]. We then applied VMProtect on the binary. We were unable to use Code Virtualizer on this real software because Code Virtualizer requires a `.map` file, which is usually generated at compile time and hence not available with malware. Table 12 shows the results of using Rotalumé with various levels of obfuscation applied to the binary.

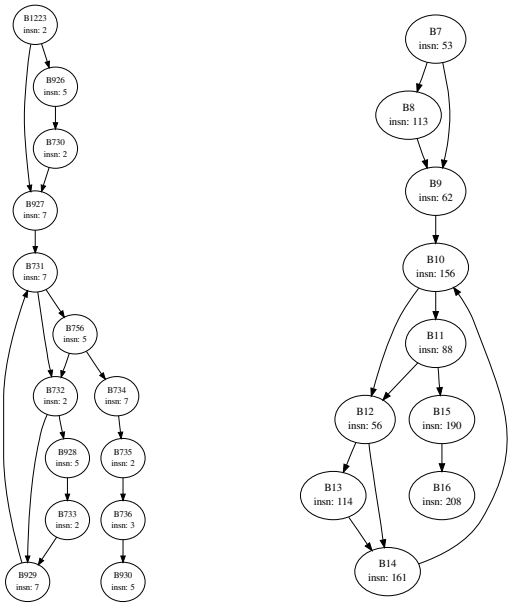
We selected one large function in the malware and used VMProtect to convert it into bytecode. The table shows that the x86 code size grows after obfuscation because the new binary additionally contains the emulator's code. Rotalumé extracted the

Table 13: Tests on `CMD.EXE` obfuscated by VMProtect

Description	Dyn. x86 CFG Inst. (BB)	Dyn. BC CFG Inst. (BB)
Original	8458 (1143)	×
1 function packed	10429 (1345)	3488 (31)
5 functions packed	10512 (1394)	12345 (103)

bytecode trace and the dynamic CFG of the obfuscated function. We compared the results with the original x86 version of the obfuscated function. Although the bytecode version had only 16 basic blocks compared to the 24 blocks of the original (not shown here; the table instead shows the size of the whole binary including the emulator), the control-flow attributes were very similar. Due to space limitations, the CFGs are not included here but may be found in a more extensive technical report [129]. From the CFGs, it seems that some basic blocks may have been combined due to code optimizations performed on the bytecode by VMProtect, but similar loops and branches were identifiable. This shows that Rotalumé indeed correctly extracted the bytecode syntax and semantics. We tested another obfuscated version of the malware where we selectively obfuscated four additional functions. In that case, the x86 code increased less substantially, and Rotalumé successfully extracted the bytecode syntax and semantics of those functions.

Finally, we tested Rotalumé after applying emulation to a legitimate program. Using `CMD.EXE`, we performed experiments similar to those for the unpacked malware. Table 13 presents the results. The bytecode CFG that we obtained contained 31 basic blocks instead of the 36 in the original function. Figure 18 shows two similar portions of the control-flow graphs of a large function of CMD.EXE. We show the original x86 code's partial CFG in Figure 18(a) and the bytecode version extracted by Rotalumé from the VMProtect obfuscated sample in Figure 18(b). The complete CFGs of the function can be found in the technical report [129]. We found that some parts



(a) Original x86 CFG

(b) Extracted bytecode CFG (VMProtect obfuscated)

Figure 18: The partial dynamic CFGs for the x86 code and the extracted bytecode of an obfuscated function in CMD.EXE

of the graphs matched perfectly, with differences in other parts likely due to code transformation and optimization differences.

6.6.3 Emulated Malware

We next evaluated Rotalumé on real malware samples that use emulation based packers. We selected samples that are packed with Themida, VMProtect, and Code Virtualizer, the three known commercial packers that use emulation. We have access to thousands of malware samples, from which we identified the ones packed using these three tools. We then applied Rotalumé to a randomly selected set of these malware samples.

Among the three obfuscators, Themida is the most widely used within our malware samples. Themida is known not to emulate the code of the original program but rather the unpacking routine. Nevertheless, we wanted to evaluate whether Rotalumé can

reverse engineer the emulator. Table 14 shows the results of Rotalumé's output on 8 randomly selected samples. We obtained the names of these samples by submitting them to VirusTotal [149] and selecting the name that was most common among the AV tools. For each sample, we gathered the execution trace when running it for 20 seconds. The table first shows the length of the x86 trace as well as the counts of instructions ("Inst.") and basic blocks ("BB") in the dynamically created CFG of the x86 code.

Our analyzer was able to detect the emulator in all cases: the table shows information for the extracted bytecode trace, and we built the control-flow graph using the extracted control-flow semantics of the bytecode language. We do not show the syntax and semantic information of the bytecode instruction set here because we found that the instruction sets consistently contain 31 instructions. However, the syntax of the instructions varied, showing that the instruction sets were highly randomized. We also manually analyzed the instruction set and observed that the semantics were very close to that from Code Virtualizer. This is not surprising given that both tools are from the same vendor [106] In all samples, the x86 CFG is very large compared to the corresponding bytecode CFG. Again, this shows that Themida was not designed to obfuscate a program completely with emulation.

We then experimented with a group of randomly selected samples that use VMProtect. We present the results of 5 samples in Table 15. Rotalumé was able to detect the emulation process in each of the samples and produced the syntactic and semantics information of the bytecode language, the bytecode trace, and the CFG. Rotalumé identified 18 bytecode instructions in the instruction set for each case. This matched the output from the synthetic test samples `synth2` and `synth3` in Table 11. Interestingly, the syntax was also the same. Like the Themida samples, these samples had very small amounts of code emulated. We conjecture that the malware authors likely had used the demo version of VMProtect, which only allows conversion of one

Table 14: Malware packed with Themida

Description	x86 Trace	Dyn. x86 CFG Inst. (BB)	BC Trace	Dyn. BC CFG Inst. (BB)
Themida5	15.3M	9753 (2156)	15232	3421 (57)
Themida1	1.4M	5961 (1156)	1339	1339 (6)
Themida3	14.8M	10211 (2125)	2142	2142 (15)
Themida7	21.4M	14205 (3529)	5171	3042 (28)
Themida8	3.5M	6011 (2125)	1534	1534 (9)
Themida11	11.1M	9021 (2925)	1784	1784 (10)
Themida13	11.4M	10211 (3194)	19642	4142 (65)
Themida14	17.3M	11492 (2877)	14219	3751 (75)

Table 15: Malware packed with VMProtect

Description	x86 Trace	Dyn. x86 CFG Inst. (BB)	BC Trace	Dyn. BC CFG Inst. (BB)
Win32.KGen.bxp	3.1M	2122 (591)	1112	1112 (9)
Win32.KillAV.ahb	1.4M	4104 (1156)	1231	1231 (12)
Graybird	131K	823 (275)	2926	1584 (18)
Win32.Klone.af*	5.0M	4263 (707)	1241	1241 (17)
Win32.Klone.af*	3.2M	4123 (484)	1149	1149 (14)

Table 16: Malware packed with Code Virtualizer

Description	x86 Trace	Dyn. x86 CFG Inst. (BB)	BC Trace	Dyn. BC CFG Inst. (BB)
Win32.Delf.Knz*	7.0M	2249 (608)	114526	10054 (343)
Win32.Delf.Knz*	15.5M	2594 (720)	234012	25221 (742)
Win32.Delf.Knz*	14.5	2531 (738)	215892	19850 (771)

function of the binary into bytecode.

We also experimented with recent malware samples that use Code Virtualizer.

Table 16 shows the results. All of the samples were identified with the same name in VirusTotal even though their program sizes and MD5 checksums varied. After analyzing these malware samples with Rotalumé, we found that unlike samples we tested with Themida and VMProtect, these samples have large portions of their code converted into bytecode. The bytecode CFGs of these programs varied significantly, showing that they may be quite different programs even though they share the same name.

6.7 Discussion

In this section, we discuss three open problems and challenges: alternative emulator designs, incomplete bytecode reconstruction, and code analysis limitations.

First, our current work assumes a decode-dispatch emulation model, thus, malware authors may implement variations or alternative approaches to emulation [80, 93, 25, 135] to evade our system. For example, our loop identification strategies of Section 6.4.3 are not directly applicable to malware emulators using a threaded approach. However, the methods of identifying the candidate bytecode regions and VPC's are still applicable. As discussed in Section 6.3.2, our approach is likewise not applicable to dynamic translation based emulation. In dynamic translation, the emulator dynamically generates new code that the program subsequently executes, thus, we expect that heuristics used by unpackers to detect unpacked code will identify the translated instructions. From the translated code, a system could trace backward to find the translation routines, and it could then utilize our methods to identify bytecode regions and the VPC. More generally, we believe that our fundamental ideas and techniques are applicable to other emulation models: by analyzing an emulator's execution trace using a given emulation model, we can identify the bytecode region and discover the syntax and semantics of the bytecode instructions. The main challenge in future research is to identify observable and discernible run-time behavior

exhibited by sophisticated emulation approaches.

Malware using decode-dispatch emulation may attempt to evade accurate analysis by targeting specific properties of our analysis. For example, since our approach expects each unique address in memory to hold only one abstract variable, an adversary may utilize the same location for different variables at different times to introduce imprecision in our analysis. Our system will put the memory reads performed using these variables into the same cluster due to the conservativeness of our analysis. If the additional data included in the cluster containing the bytecode program is used in decode or dispatch-like behavior, they may be incorrectly identified as bytecode instructions.

The second open problem is how to reconstruct complete information about the bytecode instruction syntax and semantics, so that a system can extract the entire emulated malware bytecode program. Using dynamic analysis, we extracted execution paths in the bytecode program and the syntax and semantics of the bytecode instructions used in those paths. However, the paths may not have utilized all of the possible bytecode instructions supported by the emulator, though they may be used in other execution paths of the program. A plausible approach would apply static analysis on the dispatch routine once our system has identified the emulation phases correctly. More specifically, once the dispatching method is identified, static analysis and symbolic execution may identify other execution routines and the opcodes of the bytecode instructions that invoke their dispatch. This provides the syntactic and semantic information of the bytecode instructions even though they are not part of the executed bytecode.

A subsequent open problem is utilizing the discovered syntax and semantics to completely convert bytecode to native instructions. A solution is possible only when all execution paths of the bytecode program can be explored. A potential solution is to use previous techniques employed for multi-path exploration [101] with the help of

control-flow semantics identified in the bytecode. However, emulators may be written so that specific control-flow semantics need not be supported in the bytecode language. Such is the case for VMProtect, where we have only identified unconditional branches. In such bytecode languages, the effects of conditional branches are performed in the program by dynamically computing the target address based on the condition and then using an unconditional branch to the specific target (an example was provided in Figure 15). More research is required before multi-path exploration can be applied to programs written in such languages.

Another related problem is the use of recursive emulation, which converts the emulator itself to another bytecode language and introduces an additional emulator to emulate it. The recursive step can be performed a number of times by a malware author, with size and performance increases as the limiting factors. The solution is to first apply our reverse engineering method to the malware instance, use the discovered syntax and semantics to completely convert the bytecode program into native binary code, and then apply our method (recursively) on the converted program to identify any additional emulation-like behavior.

Third, as with all program analysis tasks, reverse engineering of emulators also faces the challenges of heap analysis imprecision, limitations of loop detection, and so on. The techniques to address these problems are orthogonal to our techniques in reverse engineering.

6.8 Summary

In this chapter, we presented a new approach for automatic reverse engineering of malware emulators. We described the algorithms and techniques to extract a bytecode trace and compute the syntax and semantics of the bytecode instructions by dynamically analyzing a decode-dispatch based emulator. We developed Rotalumé, a proof-of-concept system, and evaluated it on synthetic and real programs obfuscated

with Code Virtualizer and VMProtect. The results showed that Rotalumé was able to extract bytecode traces and syntax and semantic information. By automatically defeating emulator based obfuscation, malware analysis techniques can be more robust against newer malware that employs this obfuscation. Moreover, since we presented an automated approach, the time taken to analyze malware to extract information should be significantly reduced, reducing the time required to get signatures of new malware out to the end users.

CHAPTER VII

ROBUST AND EFFICIENT MONITORING

7.1 Motivation

While robust and efficient malware analysis can lead to more accurate signatures, if the methods used by antivirus programs to utilize these signatures on target host computers remain vulnerable to attacks, the overall effectiveness of malware detection cannot improve. For example, a lot of malware employ attacks that can detect the presence of antivirus programs and either disable them completely or remove the probes that are placed in system, making the tools blind to events taking place in the system.

Today's antivirus programs run within the same host as it is supposed to secure. In order to set probes and protect itself from user level malware, these antivirus programs employ kernel level components. However, kernel-level attacks or rootkits can run in the same privilege as the operating system kernel. These attacks can modify kernel-level code or sensitive data to hide various malicious activities, change OS behavior or essentially take complete control of the system. Therefore, security tools and antivirus at the kernel level cannot have any control over their actions and cannot defend against disabling attacks.

Research as adopted virtualization as a means to provide better protect security tools. A large body of research has adopted virtual machine monitor (VMM) technology because the higher privileged hypervisor can enforce memory protections and preemptively intercept events throughout the operating system environment. A major reason for adopting virtualization is to isolate security tools from an untrusted VM by moving them to a separate trusted secure VM, and then use introspection [62, 111]

to monitor and protect the untrusted system. Approaches that passively monitor various security properties have been proposed [82, 115, 71, 74]. However, passive monitoring can only detect remnants of an already successful attack, which is not useful for a fully fledged antivirus of today. Active monitoring from outside of the untrusted VM, which has the advantage of detecting attacks earlier and preventing certain attacks from succeeding, was enabled by Lares [110]. This is achieved by placing secure hooks inside the guest VM that intercept various events and invoke the security tool residing in a separate secure VM. However, the large overhead for switching between the guest VM, the hypervisor, and the secure VM makes this approach suitable only for actively monitoring a few events that occur less frequently during system execution.

Many security approaches require the ability to monitor frequently executing events, such as host-based intrusion detection systems (IDSs) that intercept every system call throughout the system, LSM (Linux Security Module) [165] and SELinux that hook into a large number of kernel events to enforce specific security policies, or even instruction-level monitoring used by several offline analysis approaches [51]. Due to the overhead involved in out-of-VM monitoring, many such approaches either are not designed for production systems, or are not created for VM's. While keeping a monitor inside the VM can be efficient, the key challenge is to ensure at least the same level of security achieved by an out-of-VM approach.

In this chapter, we present Secure In-VM Monitoring (SIM), a general-purpose framework based on hardware virtualization features that enables security monitors residing in the same VM it is protecting to have the same level of security as residing in a separate trusted or secured VM. A security monitor in our framework retains the efficiency close to being inside the same VM by not requiring any privilege transfers when switching to the monitor for an intercepted event, and being able to access the system address space at native speed. At the same time, isolation is achieved

by putting the monitor code along with its private data in a separate hypervisor protected guest address space that can only be entered and exited through specially constructed protected gates. In other words, our system is designed in such a way that normal operation of the monitor can continue without hypervisor intervention, but any attempts to breach the security of SIM is trapped and prevented by the hypervisor. Our system design leverages Intel VT [145] hardware virtualization extensions and the virtual memory protections available in standard Intel processors.

7.2 Previous Work

7.2.1 Out-of-VM Approaches

Virtualization technology has played an important role in systems security. The security benefits gained by using virtualization has been first studied in [78, 92]. Several approaches have then been proposed to use virtual machines for providing security in production systems [110, 61, 62, 82, 115]. In general, these approaches utilize the advantage of isolation from the guest VM, and monitor some properties in the guest system to provide security. While passive monitoring has been widely used in the past, Lares [110] recently proposed the method of actively monitoring events in a guest VM. Lares provides the framework of inserting hooks inside the guest OS that can invoke a security application residing in another VM when a particular event occurs. The design of Lares enables complex and large security tools such as Antivirus programs [160] or intrusion detection systems to run on the security VM. However, the cost in communicating an event notification from one VM to another via the hypervisor makes it inappropriate for use in fine-grained active monitoring. Our approach is similar to Lares in inserting hooks inside the kernel code to invoke a monitor and provides the same security benefits. However, in our approach the monitor invocation does not require the hypervisor to be involved and the monitoring code executes with kernel-level privileges in the same guest VM. Moreover, we have

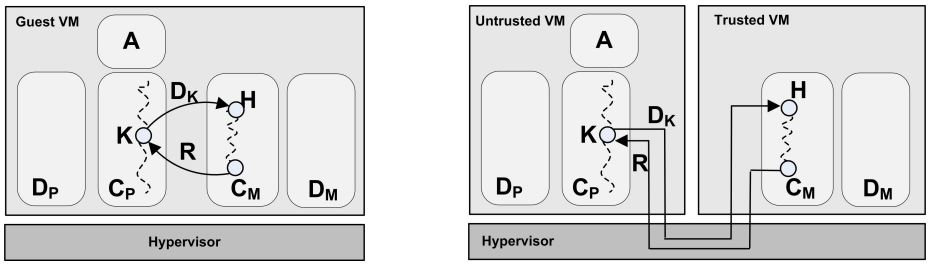


Figure 19: In-VM and Out-of-VM monitoring

the entire kernel address space visible to the monitor's address space, whereas Lares-like approaches would incur additional costs for requesting the hypervisors to map in memory belonging to another VM.

7.2.2 Hardware Virtualization

Previously most approaches used paravirtualization or software-based virtualization. The recently introduced hardware virtualization features have gained attraction in the security research community. One recent approach that uses hardware virtualization technology is Ether [51]. The goal of Ether is to a provide a transparent malware analysis environment by hiding side-effects introduced by the dynamic analyzer that monitors the run-time events at the fine-grained level. By using various features in Intel VT and carefully introducing only privileged side-effects in the guest VM, Ether can intercept accesses to these side-effects and hide them from the malware. While transparency is very important for offline malware analysis environments to prevent malware from evading analysis, it is not so for production systems. As any other security system installed in a production system, the kernel-level instrumentation and entry and exit gates of our mechanism are visible to the malware. The threat model in this scenario is the neutralization of the security mechanism instead of evasion. Our security requirements are, therefore, targeted towards this threat.

7.2.3 In-lined Monitoring Approaches

Our SIM framework can be considered as a method for enabling inlined-reference monitoring(IRM) for the kenel where the in-lined monitor is protected using hardware features. IRM has been widely adopted as a faster method of ensuring safety properties in programs by including the monitor inside the program it is monitoring. This is a far more efficient way than to have a reference monitor that resides in the kernel for user-space programs, or in the hypervisor for kernel-space programs. Traditional approaches of ensuring the integrity of the monitor itself for IRM techniques has been to ensure specific data-flow and control-flow safety properties throughout the program. For example, SFI [153] (Software Fault Isolation) is a method for having untrusted program share the same address space and provide isolation. This is achieved by rewriting specific store operations at compile time so that the address is masked in a way that it cannot write to an address region. SFI is well suited for application programs that can be modified during compile time. Achieving it for the kernel, which might have varying policies, is a hard problem. Control Flow Integrity (CFI) [14] instruments control-flow instructions with checks and their possible targets with labels at compile-time so that at run-time the checks enforce control-flow to be in the static CFG of the program. Since CFI covers all control-flow instructions, it also prevents circumvention of any of its checks. XFI [53] is an extensible fault isolation framework that provides fine-grained byte level memory access control. These features along with the protection of the XFI monitoring code is achieved by combining SFI and CFI. Finally, WIT [16] (Write Integrity Testing) provides protection from memory corruption attacks by verifying whether targets of write operations are valid by comparing with a statically and dynamically defined color table. Since write operations also encompass control-data, it provides integrity of monitoring code as well as protection from control-flow attacks without requiring CFI. Our approach of SIM provides the same isolation of the monitoring code in the kernel without having

to guarantee properties such as SFI or CFI for all kernel level code.

7.3 Efficiency and Security Requirements

A large fraction of security problems today are caused by kernel-level attacks or malicious code such as rootkits that violate some form of security in the entire system. Security tools such as anti-viruses, intrusion detection systems, and security reference monitors (e.g., SELinux) use various forms of event handling in a system in order to verify the system's security. We use the term *monitor* to represent the class of all security tools that either actively intercept events or passively analyze a system for violations of security.

If a monitor resides inside the same operating system it protects, the monitor itself can be compromised by kernel-level attacks. This problem is addressed by using virtualization. In virtual machine monitors, the hypervisor intervenes executions of the guest VM to give a virtual view of the real hardware. This is performed by utilizing the higher privilege of the hypervisor over the kernel to intercept accesses to the underlying hardware. Previous software-based virtualization utilizes a reduced privilege guest kernel, so that a higher privileged hypervisor can exist without any hardware support for virtualization. Recent processors include hardware virtualization [145] features to enable thin and light-weight virtual machine monitors. The privileged hypervisor allows various security approaches to protect access to hardware in an untrusted guest VM, and have security tools isolated from the untrusted system by placing them in a separate VM.

We illustrate two approaches of security monitoring in Figure 7.2.1. Figure 7.2.1(a) shows the model when the monitor resides in the same untrusted environment. We call this *In-VM* approach. The isolated monitor approach in a separate VM is shown in Figure 7.2.1(b), which we call *Out-of-VM* approach. At the high-level, In-VM approach provides performance and the Out-of-VM approach provides security. We

define the performance requirements based on the In-VM approach and the security requirements from the Out-of-VM approach. The goal of our work is to design a system that satisfies both the performance and security requirements.

We present a few formal notations for precision and clarity in the following discussion. Assume that a system P is being monitored by a security monitor M. The system contains code C_P and data D_P and the monitor has its code C_M along with its private data D_M. For *passive monitoring* the monitoring code C_M usually needs to analyze the system code and data and use its own data to verify the security of the system. For *active monitoring*, since an event needs to be intercepted, a set of hooks $K = \{k_1, k_2, ..., k_n\}$ are placed in the monitored system that invokes corresponding *handlers* in the set $H = \{h_1, h_2, ..., h_n\}$ contained the monitoring code C_M. A hook can pass data D_K related to the event that is gathered at the point of the hook. After the handler handles the hooked event, a response R can transfer control to any specific point in the system. It is obvious that the active monitoring model subsumes the passive monitoring case.

The overhead in executing security tools out of the guest OS is primarily due to the change in privilege levels that occurs while switching back and forth between the kernel-level and the hypervisor-level. We set the performance requirements for SIM's design to be the same as provided by In-VM approaches.

- (P1) **Fast invocation:** Invoking the monitors handler H for a hook K should not involve any privilege level changes.

- (P2) **Data read/write at native speed:** The monitor code C_M should be able to read and write any system data D_P and local data D_M at native speed, i.e., without any hypervisor intervention.

In case of in-VM monitoring, a direct control transfer to the handler code from the hook initiates the monitor. Moreover, the monitor can access all data and code

because everything is contained in the same address space. The problem of out-of-VM approaches is that both performance requirements (P1) and (P2) cannot be satisfied. First, the hypervisor is invoked when the hook K is executed to transfer control to the handler residing in another VM. Second, the hypervisor usually needs to be invoked to partially map memory belonging to the untrusted VM into an address space in the trusted VM for the out-of-VM monitor.

To state the security requirements, we consider an adversarial program A residing in the same environment as the system P. In the threat model, A runs with the highest privilege in the guest VM and therefore can directly read from, write to and execute from any memory location that is not protected by the hypervisor. To ensure the security of the monitor M, we state the security requirements:

- (S1) **Isolation of the monitor's code C_M and data D_M:** This ensures the integrity of the monitor's code and data is protected from the adversary A. Out-of-VM approaches satisfy this requirement because A does not have any means to access another guest VM.

- (S2) **Designated point for switching into C_M:** Execution should switch to the monitor only at one of the handlers in the set H. This requirement ensures that an attacker does not invoke any code in C_M other than the designated points of entry. Since the hypervisor initiates entry into the monitor, out-of-VM approaches can ensure this requirement.

- (S3) **A handler h_i is called if and only if the corresponding hook k_i executes:** This requirement has two parts - (a) If a hook k_i is reached in the monitored system, then the corresponding handler h_i must be initiated by the system. (b) an handler h_i is initiated only if the hook k_i was executed. In out-of-VM approaches, the first requirement can be satisfied by design of the handler dispatcher. The second requirement can be satisfied because the exact

`VMCalls` that initiated the hypervisor execution can be identified and checked.

- (S4) **The behavior of M is not maliciously alterable:** The execution of handlers H should not be maliciously alterable by the adversary A. First, the control-flow should not depend on any control-data that is alterable by the attacker. Second, the handlers should not need to call any dependencies that is at the control of the adversary. Third, after the handler completes, execution should return to a point that is intended by the monitor. An out-of-VM monitor can satisfy these requirements by not using any control-data contained in D_P.

None of the existing in-VM approaches can satisfy all of the security and performance requirements at the same time. First, the simple method of write-protecting the monitor's code C_M can only work for stateless monitors, which do not have any private data. A second approach may be to write protect the private data D_M using help from the hypervisor. This, however, will require the hypervisor to trap every write to verify the instruction. The performance requirement (P2) is thus not satisfied. Finally, in-lined monitoring approaches such as CFI [14] and WIT [16] can instrument each control-flow or memory write operation, usually at compile time, so that integrity checks can be enforced at run-time. A comprehensive coverage of all the required instructions needs to be performed to guarantee that the security requirements are satisfied. Such a modification of all kernel-level code is an overkill to achieve the performance requirements of a general-purpose monitoring framework that may be utilized for hooking different types of events occurring in an OS kernel. Our SIM approach is designed with all the performance and security requirements in mind.

7.4 Secure In-VM Monitoring

The goal of our Secure In-VM Monitoring framework is to enable security monitors that meet all the performance and security requirements discussed in Section 7.3. In

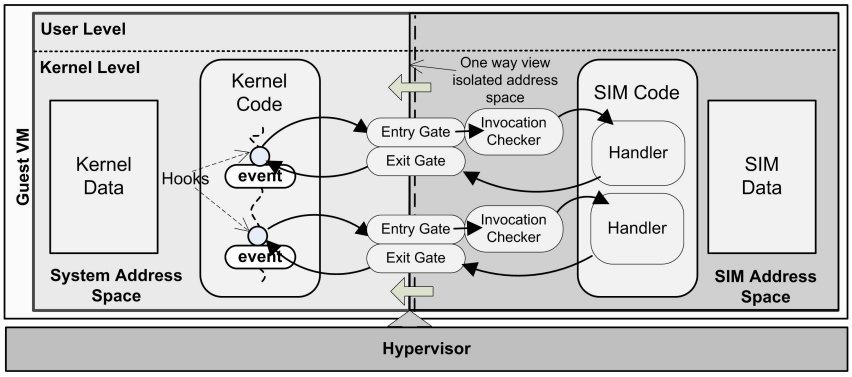


Figure 20: High-level overview of the Secure In-VM Monitoring approach

this section, we describe the design of the SIM framework.

7.4.1 Overall Design

The overall design of SIM is shown in Figure 20. The key idea of SIM is to introduce a separate hypervisor-protected virtual address space in the guest VM, which we call the *SIM Virtual Address Space*. This protected address space is used to place the security monitor. It exists in parallel to the virtual address spaces being utilized by the operating system. The virtual memory is mapped in such a way that it has a *one-way* view of the guest VM's original virtual address space. This means that the security monitor can view the address space of the operating system, but no code executing in the operating system can view the security monitor's address space. A number of *entry gates* and *exit gates* are the only code that can transfer execution between the system address space and the security monitor's address space. Hooks are placed in the kernel before specific events that transfer control to corresponding gates. The entry gate has an invocation checker module that checks who invoked the entry gate. Finally, the security monitor's code (e.g., handlers for each hook) and data are all contained in the SIM address space. Next we describe how we construct the SIM address space using paging-based virtual memory and hardware virtualization features in detail.

7.4.1.1 Protected Address Space Generation

Paging based virtual memory is generated by creating page tables that map virtual addresses to physical addresses. When an instruction is executed the current page table is used by the hardware to perform address translations. An OS creates a separate page table for each process so that it can have its own virtual memory address space and the necessary isolation can be achieved.

The memory mapping introduced by the SIM framework is shown in Figure 21. In the figure, the *process virtual address space* at the left shows the virtual address space defined by the operating system for each executing process. The virtual address space created for the SIM is shown at the right as the *SIM virtual address space*. The actual physical memory regions are shown in the middle. For now, physical memory can be considered as guest physical memory. Later, we will describe how the address translations are carried out. We have only shown the relevant kernel level addressable regions, leaving user space out of the picture. For each region in the virtual address spaces, the protection flags that are set on the relevant pages by the hypervisor are shown.

A high-level description of the contents of the process virtual address spaces are shown in Figure 21. Generally, the kernel is mapped into a fixed address range in each process's address space. We call this address range the *system address space*. Since we are primarily focusing on kernel level monitoring, we illustrate the contents of this address range in the figure. We denote any code and data contained in the system address space as *kernel code* and *kernel data*. All pages containing kernel code will have read and execute privileges, but we assume that the kernel code can be write protected, especially places where hooks are placed. The data regions will have all access rights. In general, the dynamically loaded kernel-level code would be maintained in the kernel data region. In our framework, we introduce the entry and exit gates into the system address space. As mentioned earlier, the gates are used

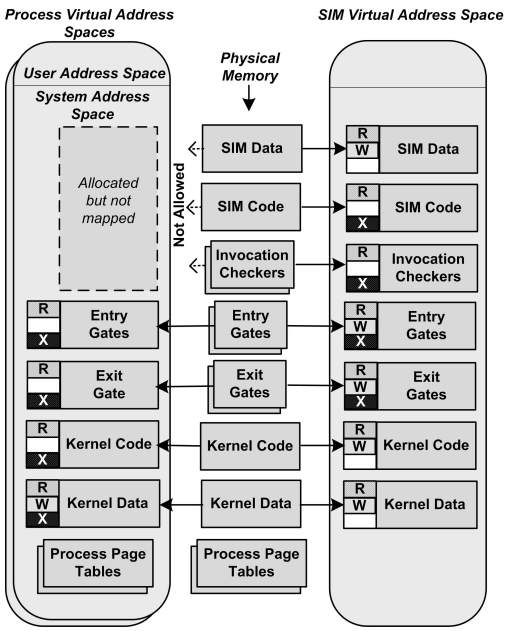


Figure 21: Virtual Memory Mapping of SIM approach.

to perform transitions between the system address space and the SIM address space. Since the gates include code, they are set with execute permissions but are made read only so that they cannot be modified from with the guest VM.

The SIM address space includes the security monitor's code (*SIM code*) and data (*SIM data*). Besides the security monitor, the SIM address space contains all the contents of the system address space that are mapped in. However, some of the permissions are set differently. The kernel code and data regions do not have execute permissions. This means that while execution is within the SIM address space, no code mapped in from the system address space will be executable. The invocation checking modules are also contained only in the SIM address space and have execution privileges.

Since the system address space contents are mapped into the SIM address space, an important requirement for the mapping to work is to ensure that other (the additional) regions in the SIM address space (i.e., the SIM code, SIM data and the invocation checker regions) do not overlap with the mapped in regions from the system address

space. There are two methods to achieve it. First, the virtual address range that is used for user programs may be used for allocating the SIM regions. This approach is suitable for security monitors that will be primarily used to monitor kernel level code. Second, we can use OS provided functionality to allocate memory from the system address space. Once allocated, any legitimate code (including the OS itself) should not attempt to use this memory region in the system address space. Possible attacks may arise, which are discussed in Section 7.4.3.

Since the SIM address space contains all kernel code, data and also the SIM data in its address space, the instructions as part of the security monitor can access these regions in native speed. This satisfies the performance requirement (P2). The memory mapping method we have introduced satisfies the isolation security requirement (S1) by having the SIM code and data regions in a separate SIM address space. Any kernel-level instruction executing in the guest OS will utilize the system address space, which do not include these regions. Although any kernel-level code executing in the OS environment has full freedom to change the process virtual memory mappings because they are mapped into the system address area, they cannot modify or alter the SIM address space. By design, the SIM page table is neither included in the system address space, nor the SIM address space. In Section 7.4.1.2, we will explain the reason behind it and how we achieve it.

7.4.1.2 Switching Address Spaces

In the Intel x86 processors, the `CR3` register contains the physical address of the root of the current page table data structure In the two level paging mechanism supported in the IA-32 architecture, the root of the page table structure is called the *page directory* [70]. As part of the process context switching mechanism, the content of the `CR3` register is updated by the kernel to point to appropriate page table structures used by the current process. Although the kernel of the OS mainly

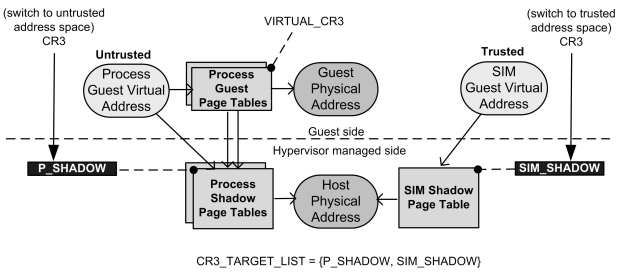


Figure 22: Switching between the untrusted and trusted address space without hypervisor intervention.

maintains the valid CR3 values to switch among processes, any code executing with the kernel-level privilege can modify the CR3 register to point to a new page table. However, to ensure the correct operation of the operating system, kernel code needs to see its expected CR3 values.

In virtual machines, the page tables in the guest VM are not used for translating virtual addresses to physical addresses because the physical memory that needs to be translated to is on the host, which is maintained and shared among various VM's by the hypervisor. The hypervisor takes complete control over the guest OS memory management by intercepting all accesses to the CR3 register. The guest physical memory then only becomes an abstraction utilized by the hypervisor for maintaining correct mapping to the host physical address. *Shadow Page Tables* are used by the hypervisor to map guest virtual to host physical memory [23, 86, 10] and different mechanisms are used in different VMM implementations to maintain consistency among the guest page tables and the shadow page tables. The hypervisor gives the guest OS the illusion that its designated page tables are being used.

Since our mechanism requires the switching of address spaces, we need to modify the `CR3` register contents directly. However, the modifications to the `CR3` register by the guest VM is trapped by the hypervisor. The challenge is to bypass the hypervisor invocation, so that the performance requirement (P1) can be satisfied. For this reason, we utilize a hardware virtualization feature available in Intel VT. By default all

accesses to the guest CR3 register by the guest VM causes a VMExit, which is a switch from the guest to the hypervisor. Intel VT contains a feature that it does not trigger a VMExit if the CR3 is switched to one of the page table structure's root addresses in a list (`CR3_TARGET_LIST`) maintained by the hypervisor [70]. The number of values this list can store varies from model to model, but the Core 2 Duo and Core 2 Quad processors support a maximum of 4 trusted CR3 values to be added in the `CR3_TARGET_LIST`.

The guest OS provides the addresses of guest page directories in the CR3 register, and the correct execution of the guest VM is ensured by the hypervisor changing them to the appropriate shadow page directories instead. However, if we bypass the hypervisor while switching CR3 values, we need to directly switch between the shadow page directories. Figure 22 illustrates how the switching is performed by updating CR3 register. Besides the hypervisor maintained shadow page table structures, we introduce an additional specialized shadow page table, which we call *SIM shadow page table*. The page table converts virtual addresses in the SIM address space to host physical addresses. Since it is directly maintained in the hypervisor and the security monitor need not manage its virtual memory, we eliminate the need for any guest level page table for the SIM address space. The root of the SIM shadow page table structure is the *SIM shadow page directory*, which we designate as `SIM_SHADOW`. We also designate the physical address of the current shadow page directory maintained by the hypervisor as `P_SHADOW`. Switching between the process address space and the SIM address space requires to directly modify the CR3 register and load the value of `SIM_SHADOW` or `P_SHADOW` after already adding them to the `CR3_TARGET_LIST`. This ensures the correct operation of the code in the guest VM when the hypervisor is not involved. The entry and exit gates described in Section 7.4.1.3 perform this switching, and the rest of the design of the SIM framework ensures that the switching is transparent to the guest OS.

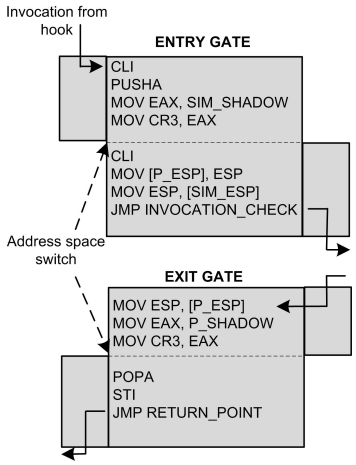


Figure 23: Entry and exit gates

7.4.1.3 Entry and Exit Gate Construction

The entry and exit gates are the only regions that are mapped into both the system address space and the SIM address space in pages having executable privilege. This ensures that a transfer between the address spaces can only happen through code contained in these pages. Moreover, since these pages are write-protected by the hypervisor, its contents cannot be modified by any in-guest code. The contents of the entry and exit gates are shown in Figure 23.

As mentioned earlier, each hook and handler have a pair of corresponding entry and exit gates. The task of an entry gate is to first set the CR3 register with the physical address of the SIM shadow page directory, or `SIM_SHADOW`. This switches the entry into the SIM address space. Since the CR3 register cannot be directly loaded with data, the value of `SIM_SHADOW` first needs to be moved to a general purpose register. For this reason, we need to save all register values on the stack, so that the security monitor can access register contents at the point when the hook was reached. Even though the register contents are saved on the stack on the system address space, since interrupts are disabled by the entry gate already, an attacker will not be able to regain execution and modify the values before entry into the SIM address space. Once in the SIM address space, the next task is to switch the stack to

a region contained in the SIM data region by modifying the ESP register. The stack switching is necessary, so that code executing in the SIM address space does not use a stack provided by the untrusted guest kernel-level code. Otherwise, an attacker can select an address in the form of the stack pointer that may overwrite parts of the SIM data region once in the SIM address space. Finally, control is transferred to the invocation checker routine to verify where the entry gate was invoked (discussed in Section 7.4.1.4). Notice that the first instruction executed in the gate is the CLI to stop interrupts from executing. This guarantees that execution is not diverted to somewhere else due to interrupts. The reason for executing the same CLI instruction again after entering the SIM address space is discussed in Section 7.4.3.

The exit gate performs the transfer out of the SIM address space into the process address space. First the stack is switched back to the stack address saved during entry. To make the address space switch, the CR3 register is loaded with the address in P_SHADOW, which is the physical address of the shadow page table root. The hypervisor may be using multiple process shadow page tables and switching between them as necessary. To ensure correct system state, the value of P_SHADOW should be equal to the address of shadow page directory being used by the hypervisor prior to entering the SIM address space. Querying the hypervisor for the correct value during monitor invocation violates the performance requirement (P1). We take the approach of making the hypervisor update the value of P_SHADOW used in the exit gates when it switches from one process shadow page table to another. Having the value of P_SHADOW as an immediate operand in every exit gate would require the hypervisor to perform several memory updates. Instead, storing it as a variable in the SIM data region requires only one memory update by the hypervisor at the time of shadow page table switches. At the end of the exit gate, the interrupt flag is cleared to enable interrupts again, and then execution is transferred to a designated point usually immediately after the hooked location. The exit gates have write permissions in the

SIM address space, enabling the security monitor to control where the execution is transferred back.

The entry gates are the only way to enter the SIM address space, and they first transfer control to the corresponding invocation checking routine, which then calls a handler routine. By doing so, we ensure the security requirement (S2). Moreover, the "if" part of the requirement (S3a) is satisfied because, when a hook is executed, the corresponding handler is invoked. Additional variations of attacks are also handled by our design and are discussed in Section 7.4.3.

7.4.1.4 Checking Invocation Points

To satisfy the security requirement (S3b), once the SIM address space is entered through one of the entry gates, the invocation of the gate needs to be checked to ensure that it was from the only hook that is allowed to call the gate. The challenge is that since the entry gate is visible to the guest OS's system address space, a branch instruction can jump to this location from anywhere within the system address space. Moreover, we cannot rely on call instructions and checking the call stack because they are within the system address space and as such the information cannot be trusted. We utilize a hardware debugging feature available in the Intel processors after Pentium 4 to check the invocation points. This feature, which is called *last branch recording*(LBR) [70], stores the sources and targets of the most recently occurred branch instructions in some specific processor registers.

The last branch recording feature is activated by setting `LBR` flag in the `IA32_DEBUGCTL` MSR. Once set, the processor records a running trace of a fixed number of last branches executed in a circular queue. For each of the branches, the IP (instruction pointer) at the point of the branch instruction and its target address are stored as pairs. The number of these pairs stored in the LBR queue varies among the x86 processor families. However, all families of processors since Pentium 4 record information

about a minimum of four last branches taken. These values can be read from the MSR registers MSR_LASTBRANCH_k_FROM_IP and the MSR_LASTBRANCH_k_TO_IP where k is a number from 0 to 3.

We check the branch that transferred execution to the entry gate using the LBR information. In the invocation checking routine, the second most recent branch is the one that was used to invoke the entry gate. We check that the source of the branch corresponds to the hook that is supposed to call the entry gate. Although the target of the branch instruction is also available, we do not need to verify it if the source matches. Our design also mitigates possible attacks that may jump into the middle of the entry gate and try to divert execution before invocation checking routine is initiated. This is discussed in Section 7.4.3.

A conceivable attack may be an attempt to modify these MSR registers in order to bypass the invocation checks. We need to stop malicious modifications to these MSR, but at the same time ensure that performance requirement is not violated. With Intel VT, read and write accesses to MSR registers can selectively cause VMExits by setting the MSR read bitmap and MSR write bitmap, respectively. Using this feature, we set the bitmasks in such a way that write attempts to the IA32_DEBUGCTL MSR and the LBR MSRs are intercepted by the hypervisor but read attempts are not. Since the invocation checking routine only needs to read the MSRs, performance is not affected.

7.4.2 Security Monitor Functionality

One of the important aspects of our design is to ensure that the security monitor code does not rely on any code from any untrusted region. Therefore, the security monitor code needs to be completely self-contained. This means that all necessary library routines need to be statically linked with the code and the monitor cannot call any kernel functions. From design, mapping the kernel code and data with non-execute

privileges ensure that even any accidental execution of untrusted code does not occur in the trusted address space (because execution on non-execute code and data will result in software exceptions). Any software exceptions occurring while in the SIM address space is handled by code residing in SIM. Moreover, the entry and exit from the SIM address space can be considered an atomic execution from the perspective of the untrusted guest OS. While the hypervisor will receive and handle interrupts on the guest OS's behalf, they are not notified to the guest VM while the interrupts are disabled in the guest VM. Disabling interrupts before entering and after exit ensures that interrupts do not divert the intended execution path of the security monitor, which guarantees the security requirement (S4). Even without using the code of the guest OS, the same functionality provided by an out-of-VM approach can be achieved in our design.

First, by not allowing kernel functions to be called, the security monitor needs to traverse and parse the data structures in the kernel address space in order to extract necessary information required for enforcing or verifying security state of the untrusted region. However, this is the same semantic gap that exists while using introspection to analyze data structures of the untrusted guest VM from another trusted guest VM. The method of identifying and parsing data structures used in existing out-of-VM approaches can therefore be ported to our in-VM approach with a few modifications.

Second, a security monitor may need to perform accesses to hardware or perform I/O for usability purposes besides handling the events in the untrusted guest OS. Theoretically, it may be possible to replicate the relevant guest OS functionality inside the SIM address space. However, accessing hardware directly may interfere with the guest OS. We take a different step in our design. Since the SIM address space can be trusted, we allow a layer to be defined that communicates with the hypervisor for OS-like functionality through hypercalls. This layer, which we call

the *SIM API*, can provide functionalities such as memory management, disk access, file access, additional I/O, etc. This layer can be developed as a library that can be statically or dynamically linked with the security monitoring code based on the implementation. The handling of the SIM API can be performed in the hypervisor or it may be performed by another trusted guest VM. Since the security monitor can be designed to use such functionality less often than handling events in the untrusted guest kernel (e.g., buffering data), the cost of hypervisor invocation can be kept low even for fine-grained monitoring.

7.4.3 Security Analysis

We first summarize how the our design has met all the security requirements stated in Section 7.3 without sacrificing any of the performance requirements. SIM satisfies the security requirement (S1) by using the hypervisor to not allow the monitor code and data to be mappable to any untrusted address space in the guest VM. The monitor remains in a completely isolated trusted address space isolated from the attacker. The requirement (S2) is satisfied because by design, the only method to enter the trusted address space from the untrusted one is via the entry gates. Since each hook invokes a corresponding entry gate, which eventually calls a corresponding handler, and each invoker of the entry gate is checked by the invocation checking routine, the requirement (S3) is satisfied. Finally, by not allowing any code from the untrusted domain to be executable in the trusted address space, and by design not allowing the monitor to call into the untrusted kernel, we ensure that the security requirement (S4) is met. In the rest of this section, we discuss a few variations of attacks that are also handled by the design.

One important design consideration is to stop attacks that may divert the control-flow of the security monitor by modifying the control-data. Since any data in the untrusted region are completely at the hands of the adversary according to our threat

model, it is important that, by design, the security monitor stores all control-data that it needs in the SIM data region. Ensuring this does not in any way reduce the functionality of the security monitor.

An attacker may directly call the entry gate and skip the first interrupt disabling instruction with an intention of keeping interrupts enabled after entering the SIM address space and before the invocation routine is executed, causing undefined behavior in case the interrupts are not handled properly. Having another `CLI` instruction at the beginning of the entry into the SIM solves this issue.

Since the entry gate is visible to the guest OS kernel-level code, the value of the SIM_SHADOW is revealed to the attacker. Although this value is known to the attacker, it cannot be used to switch into the SIM virtual address space from the attacker's kernel-level code. This is because, once the instruction that loads the CR3 register with the SIM shadow page directory's address is executed, the address space is switched immediately. The instruction immediately following the load instruction is from the new address space. Since the attackers code will not have execute privilege in the SIM address space, an exception will be generated. One possible modification of the attack is to allocate a page in the system address space that precedes immediately before a page that is used inside the SIM address space. If the instruction for setting the CR3 is the last instruction in the page, the next instruction executed will be a valid address inside the SIM address space. In order to defeat this attack, we ensure that each page whose immediate previous page does not contain code or have the executable privilege if it is allocated in the system address space. This ensures that such illegal entry into the SIM address space can be prevented.

7.5 Implementation

We have implemented a prototype of the SIM framework We used KVM (Kernel Virtual Machine) [86] for implementing the hypervisor component of the SIM framework

on a Linux host with 32-bit Ubuntu distribution. The implementation was done on a system with Intel Core 2 Quad Q6600 processor, which has Intel VT support. We targeted Windows XP SP2 as the guest OS. To generate the SIM address space and load a security monitor into it, we rely on an *initialization phase*. Then, during the system runtime, the hypervisor based component provides memory protection, updates exit gates and handles VM calls that are relevant to the SIM API. We describe the initialization phase in Section 7.5.1 and the execution phase in Section 7.5.2.

7.5.1 Initialization Phase

The initialization phase of our system is initiated by a guest VM component implemented as a Windows driver that is executed after a clean boot when the guest OS can be considered to be in a trusted state. The primary task of the initialization driver is to allocate guest virtual memory address space for placing the entry and exit gates based on the hooks required, initiate creation of SIM virtual address space, initiate the loading of the security monitor into the address space, and finally the creation of entry gates, exit gates and invocation checking routines. The initialization driver communicates with the hypervisor counterpart of SIM using a hypercalls. We use the `VMCALL` instruction of Intel VT for the hypercall and use the four general purpose x86 registers to store arguments.

The first task is to reserve virtual address ranges in the system address space for use in entry and exit gate creation. Since we need to guarantee that the normal operation of the OS and legitimate applications do not attempt to utilize the reserved address ranges, we rely on the guest OS to allocate virtual address space. The driver allocates contiguous kernel-level memory from the non-paged pool by using the `MmAllocateContiguousMemory` kernel function. The function returns the virtual address pointing to the starting of this allocated memory region. Since the function allocates memory from the Windows non-paged pool, it is guaranteed by the OS to

be never paged out. In other words, the pages are mapped to guest physical frames that are not used until they are freed. Since the memory is already allocated, any legitimate application will not try to utilize this address space. The allocated virtual address space region is informed to the hypervisor component using a predefined hypercall notifying the starting address and the size of the allocated region. During execution, our system checks for any malicious attempts to utilize this address space or changes in memory mapping.

The next step is the creation of the SIM virtual address space by the hypervisor component of the SIM framework. Once the hypervisor is informed about the memory allocation, the SIM shadow page table structure is created. In the 32-bit implementation of KVM, we noticed that KVM's own shadow page tables are implemented not as regular two-level 32-bit page tables, but as three-level 36-bit PAE (Physical Address Extensions) [70] page table structures. The Intel processors support this type of page table structure to handle more than 4GB of physical memory. To keep implementation elegant, one of our goals was not to make extensive modifications to KVM's MMU (Memory Management Unit) code, which mainly handles the shadow page table algorithms. Rather than utilizing the MMU code of KVM, we wrote our own code to create, maintain and update our SIM shadow page table. Since we need to switch between the same type of shadow page tables, we also implemented the SIM shadow page table as a PAE 36-bit page table structure. The usage of the PAE page table also enabled us to set NX-bits on pages, even though the 32-bit page tables used by the guest OS do not support this feature. During generation of the shadow page table, the system address space is traversed in the current process page table and mappings of all relevant entries are added to the SIM shadow page table with appropriate privileges.

In our current prototype implementation, the method we used for loading a security monitor application into the SIM address space is to load the application as part

of the kernel driver in the system address space and then inform the hypervisor to place it into the SIM address space. The driver performs a hypercall with the starting address of the monitor code and its size. This hypercall enables the hypervisor component to allocate entries in the SIM shadow page table structure and map the virtual address range to newly allocated host physical memory that holds the monitor code as the SIM code region. We keep the same virtual address range so that no address relocation needs to be performed. The allocated host physical memory is intentionally not correlated with any guest physical memory, making it impossible for any guest page table to try to map these regions. In the same manner, the monitor data region is mapped into the SIM address space as SIM data. In our current prototype, we only support static data regions. Supporting dynamic data regions requires additional engineering effort and is orthogonal to the current design of our framework with a focus on performance and security.

The final task is to create the relevant routines to perform switching into the SIM address space. The security application requires hooking into the kernel for invoking its handlers. Any form of hooking can be utilized by the handler, and the method of hooking is orthogonal to our framework. For each hook and handler, a hypercall is performed by the driver to inform the hypervisor about the hook instruction, the handler's address and the address where to return execution to after the handler executes. For each received hypercall, the hypervisor component of our framework generates an entry gate, an invocation check routine, and an exit gate. The invocation checking routine is modified to verify the invocation instruction address to be the hook instruction address provided with the hypercall. A jump instruction is placed at the end invocation routine to jump to the provided handler. The exit gate code is also modified to return execution to the specified address. The address of the entry gate is returned, so that the driver can modify the hook to divert execution to the entry gate.

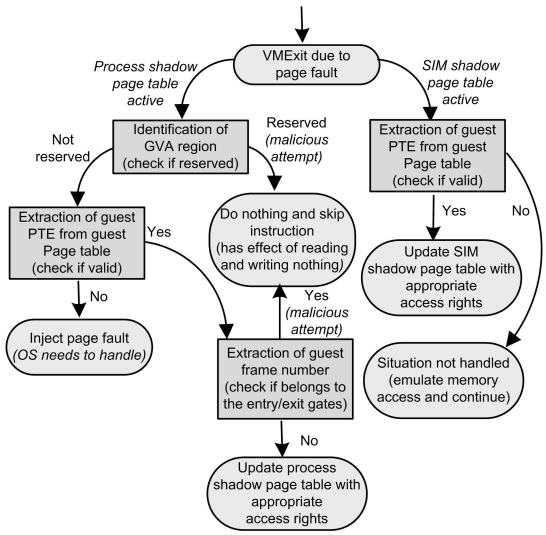


Figure 24: Run-time memory protection flowchart

7.5.2 Run-time Memory Protection

At run time, attacks may attempt to breach the security enforced by the SIM framework by changing the virtual memory mapping in the system address space, or introducing a new virtual memory mapping. Rather than checking the guest page table entries for security violations whenever the guest CR3 is switched to a new page directory address, we ensure the security of the SIM framework by verifying memory protection whenever a guest page table entry is propagated to the shadow page table maintained by the hypervisor. Figure 24 illustrates the memory protection in our prototype.

Whenever a page fault occurs for the shadow page tables being used, a VMExit is called and trapped by KVM. The guest virtual address (GVA) for which the page fault occurred is identified by reading the guest CR2 register. We first check whether a process shadow page table or the SIM shadow page table is active. If the process shadow page table is active, we check whether the virtual address is in a reserved region (the address space reserved for SIM). If it is, it indicates a malicious memory access because this region is already allocated by the guest OS. We simply skip the instruction without doing anything, emulating the effect as if reading and writing

succeeded. If the memory region is not reserved, we extract the guest page table entry (PTE) by traversing the current guest page table. If the PTE is not valid, we inject a page-fault into the guest, causing the OS to handle a regular page-fault situation. If it is valid, we then check whether the guest PTE is illegally trying to attempt to map the faulting GVA to guest physical memory containing entry or exit gates. In case of an illegal mapping attempt, we skip the instruction without doing anything. In case of a valid mapping, we update the process shadow page table. We ensure that the appropriate memory protection bits, such as write-protection, are enabled for PTE's updated in the shadow page table.

If the page-fault occurs while the SIM shadow page table is active, it must be due to accessing the kernel code, kernel data or user space regions in SIM since other SIM-specific code and data were already mapped in the initialization phase. Therefore, we traverse the current guest page table's system address space to search for the page table entry (PTE) corresponding to the faulting address. If the PTE entry is valid, in other words, the page is mapped to a valid guest frame number, we identify the host frame number and update the SIM shadow page table by inserting a corresponding PTE. However, we ensure that the mapped page does not have the execution privilege and has complete read-write access. If the PTE is invalid or is indicated as not present, it means that the guest OS possibly has paged-out the region. We currently cannot handle this case because it requires the untrusted guest OS to handle the page-fault. This is a violation of the security requirement (S4) because an attacker may change the behavior of guest page fault handler or insert its own. Our approach is to skip the instruction as if it succeeded without any effect. This problem can be overcome by having the hooking routine access these required memory regions before invoking the entry gate, causing all potentially paged-out frames to be paged-in before entering the SIM address space.

7.6 Experimental Evaluation

7.6.1 Monitor Invocation Overhead

We performed micro benchmarks to compare the overhead in invoking an SIM monitor with an out-of-VM one. We measured the time required to switch to the monitor code from a hook and then switch back. For this reason, we implemented null event handlers that return immediately without performing any useful task.

For measuring our SIM framework, we implemented a security monitor whose handler only calls the corresponding exit gate. Hence, we could measure the monitoring overhead by invoking the entry gate, which caused a transition from the system address space to the SIM address space and back.

For measuring the out-of-VM overhead, we developed a simple inter-VM communication mechanism that enables a driver residing in the untrusted VM to invoke code of a driver placed in the trusted VM. Due to space constraints, we omit the implementation details. We measured the time taken from the hook being executed, initiating the communication from the driver in the untrusted VM to receiving execution back with a response from the trusted VM.

Table 17 shows the results of our micro benchmarks. Invoking the SIM monitor is almost 11 times faster than the out-of-VM monitor. We acknowledge that the overall performance gain will also depend on the processing time required for each invocation. A monitor that is frequently invoked with a small amount of processing in each invocation (i.e., fine-grained monitoring) should benefit from the SIM framework. In cases where the monitor is less frequently invoked and a large amount of processing is performed in each invocation, our SIM approach may have less performance advantage. However, this experiment only shows the switching overhead between the monitored system and the security monitor, leaving out additional performance penalties out-of-VM approaches may face due to external introspection costs. The experimental results in the Section 7.6.2.1 will show a more practical comparison of

Table 17: Monitor Invocation Overhead Comparison

Monitor type	Avg. time (μ sec)	Std. dev. (μ sec)
SIM approach	0.469	0.051
Out-of-VM approach	5.055	0.132

performance overhead by taking into account both switching and introspection costs for a real-world security monitoring application.

7.6.2 Security Application Case Studies

We developed two security applications using the SIM framework to perform a more elaborate and practical evaluation of the performance advantages of the SIM approach over an out-of-VM approach. We first developed a process creation monitor using our SIM framework and the out-of-VM approach. The monitor intercepts the creation of each process in the untrusted domain and allows execution if the name of the process is contained in a white-list. Micro benchmarks on this application will not only take the monitor invocation overhead into account, but also consider any overhead introduced due to introspection performed by the out-of-VM monitor. Section 7.6.2.1 shows performance evaluation using this application. Although micro benchmark tests allow comparison of overhead introduced for monitoring individual events, it does not show the impact of monitoring on an overall system where several events are monitored continuously. Since system call events occur frequently, we developed a system call monitoring tool and performed macro benchmarks on a number of representative applications running in the monitored guest VM. We provide evaluation of SIM using this application in Section 7.6.2.2.

7.6.2.1 Process Creation Monitor

The process execution monitor that we developed works similarly to the application presented in Lares [110]. We hooked the `NtCreateSection` system call in Windows

by modifying the *System Service Descriptor Table* (SSDT). This system call is always invoked when a new process is created. The seventh argument to the system call is the handle related to the executable file, which lets the executable's name to be determined. Although the identification of the executable's name from the handle is straightforward using Windows kernel functions, we had to traverse the kernel data structures directly to extract this information in both the SIM-based and out-of-VM approach-based security monitors.

The security monitoring tool first extracts the `ETHREAD` structure of the current thread of the current process by traversing the *Thread Information Block* (TIB) pointed to by the `FS` segment selector. The `ETHREAD` structure is traversed to extract the `EPROCESS` structure related to the current executing process that performed the system call. We then traverse the object handle data structures to identify the object relevant to the object handle sent as an argument to the `NtCreateSection` system call. Once this object is identified as a file, the path name is extracted and compared with a list of allowed process images predefined in the tool. Depending on whether it is allowed or not, the original system call handler is called or skipped with a failure indication, respectively.

The SIM-based tool was written to directly access the system address space to traverse kernel data structures. However, the out-of-VM version requires introspection like functionality. Since KVM does not come with an introspection API, we implemented a limited form of introspection functionality in KVM ourselves. A driver in a guest VM calls a hypercall to map a page in its current system address space to the physical frame corresponding to a virtual address in the system address space of another VM. We were able to support this functionality in the hypervisor by simply updating the shadow page table corresponding to the guest VM's current guest page table with the requested mapping.

The performance micro benchmark results of the process execution monitor are

Table 18: Process creation monitor performance results

Monitor type	Average time (μ sec)	Relative overhead (%)
Traditional	3.487	×
Out-of-VM approach	28.039	690.5%
SIM approach	3.967	13.7%

provided in Table 18. We compared the performances of the SIM-based and the out-of-VM monitor with a *traditional* monitor that resides in the guest VM without any isolation. The out-of-VM version has almost 700% overhead over the traditional one. Our analysis revealed that the additional overhead beyond the invocation cost was due to additional hypercalls made for introspection purposes. With our SIM-based implementation, we only incurred an overhead of 13.7%, which is magnitudes lower than the out-of-VM approach. This overhead is primarily due to the cost of switching address spaces, which also invalidates the processor TLB cache, requiring additional memory accesses to page table. The results show that, besides reducing switching costs (as shown in Section 7.6.1), our SIM approach provides significant gains in performance in practical settings by providing the ability to access the monitored kernel data structure directly rather than the costly introspection methods used by out-of-VM approaches.

7.6.2.2 System Call Tracing

To evaluate the performance advantages of using SIM when frequent invocations are necessary, we developed a system call monitoring security tool and measured the run-time overhead on a system using a set of macrobenchmarks. To hook every system call occurring in the guest VM, we modified the `SYSENTER_EIP` MSR. In our prototype, we did not perform any specific system call inspection but read the stack to emulate the processing time a system call check would require.

For our SIM-based tool, we modified the MSR to point to code snippet that

Table 19: System call tracing macrobenchmarks

Benchmark	Bare Bare	Out-of-VM approach	SIM approach
Memory Latency	10.42 MAcc/s	84.58%	7.97%
HTML Render	1.12 pg/s	52.42%	5.83%
File Compress	3.4 MB/s	3.97%	0.59%
File Encrypt	20.56 MB/s	7.85%	0.89%
File Decrypt	78.21 MB/s	2.53%	0.45%
HDD	15.29 MB/s	41.68%	3.74%
Text Edit	82.73 pg/s	128.84%	9.64%
Average	×	46.10%	4.15%

disables interrupts, jumps to the entry gate for the system call handler inside the monitor and upon return from the exit gate, jumps to the original system call handler. The system call handler code in the SIM extracts the userland stack address from register values saved on the stack by the entry gate. The handler stores the system call number together with a fixed number of arguments on the stack in a circular queue statically allocated in the SIM data region. After every 10,000 system calls, a hypercall is made to output a trace of the system calls via the SIM APIs.

In the out-of-VM approach, we pointed the `SYSENTER_EIP` MSR to a code snippet that performs a hypercall with the syscall number and the current userland stack address (that contains the arguments of the system call). The hypercall initiates communication with a driver in another VM that performs introspection on the userland stack to retrieve the arguments. The driver places the system call information in the circular queue which is sent to a kernel log after every 10,000 calls.

For performing the macrobenchmarks, we chose PCMark 05 [112] as the benchmarks suite. We did not include any graphics or audio tests because the drivers in KVM mostly emulate these hardware components. We first ran the benchmark in a guest VM without any system call monitoring tool installed. Then we performed the same tests with our SIM-based monitoring tool and the out-of-VM tool. The results are shown in Table 19. Regardless of applications, the SIM approach had overhead

a magnitude less than the out-of-VM approach. The overhead varied significantly among the applications for both the SIM and the out-of-VM versions, which was primarily due to the varying rate of system call invocations. The average overhead introduced by SIM was 4.15% compared to 46.10% of the out-of-VM version.

Since we did not parse the system call arguments based on their types, the amount of introspection required by the out-of-VM approach was fairly limited. Only one introspection VMCall was required per system call. In a full-fledged system call tracer that utilizes the argument type information, the overhead for introspection will have a significant impact on performance and our SIM approach should then show an even larger degree of performance advantage over out-of-VM approaches.

7.7 Summary

The general-purpose Secure In-VM Monitoring framework that we presented in this chapter provides the same security guarantees of out-of-VM monitoring and yet incurs similar low performance overhead of in-VM monitoring. We described the design of SIM and presented a comprehensive security analysis. We have implemented a prototype SIM on KVM, and performed benchmarks on two representative security monitoring applications to compare the SIM approach with a typical out-of-VM one. Our microbenchmark results show that the SIM framework can reduce monitoring overhead by almost 11 times if only monitor invocation time is considered. The microbenchmarks on an introspection-heavy security application shows that SIM only introduces an overhead of 13.7% compared to 690.5% for an out-of-VM approach. In terms of the overall overhead on a system with a large number of event hooks and hence frequent invocations of the monitor, SIM keeps the overall overhead below 10% while an out-of-VM approach has overhead as high as 128%.

The goal of having both a robust and efficient system monitoring mechanism for an

antivirus or security tool is achieved by SIM. Since the security equivalent to an out-of-VM approach is provided, any unknown malware should not be able to modify the code and data of an antivirus program residing in SIM. Probes placed in the operating system environment can also be protected using the help of the hypervisor. Finally, although the same efficiency of an kernel module residing in the same environment without a protection like SIM is not achieved, the cost of switching to the antivirus is magnitudes less than out-of-VM approaches, and should make developing more robust and protected antivirus programs possible.

CHAPTER VIII

CONCLUSION AND FUTURE WORK

8.1 Summary

As computers are becoming more ubiquitous, the potential threat of malware continuous to grow. The current state of malware detection is showing that many antivirus products and host-based security tools do not provide adequate security to protect a computer from being infected. In this dissertation, we have addressed this important issue and focused on several problems that can help antivirus and security tools be more successful in detecting today's malware. While there can be many different approaches to detecting malware, utilizing various signature-based or behavior based models, they all can benefit from advances in two aspects in the malware detection life-cycle that are orthogonal to their design. One is the approach of extracting information from malware to build the detection model, which is called malware analysis. And the other is the approach in which an antivirus utilizes the detection models and monitoring events or code in the host to detect malware. We have addressed problems in both of these areas with a goal to improve malware detection.

First, we provided methods that can make various malware analysis approaches more efficient and robust against malware. We focused on both static analysis and dynamic analysis to make them more robust against anti-analysis attacks. The contributions in this area are provided below:

- We introduced Eureka, a framework that is targeted to enable static analysis on malware by defeating single-pass and multi-pass packing obfuscations, as well as API obfuscation attacks. When developed, our evaluation showed that

it successfully unpacked 95% of daily occurring real-world packed malware binaries. However, we currently anticipate that due to more popularity of new obfuscations, this has gone down. Nevertheless, our method efficiently generates unpacked malware for further static analysis.

- In order to improve the robustness of dynamic analysis, we presented a formal framework for describing program execution and analyzing the requirements for transparent malware analysis. We then presented Ether, an external, transparent malware analyzer that operates using hardware virtualization extensions to offer both fine- (single instruction) and coarse-(system call) granularity tracing.

The next part of the thesis focused on obfuscation techniques that can directly affect both static and dynamic analysis approaches. While most obfuscation techniques target static analysis, we focused on obfuscation techniques that can affect dynamic analysis as well. We first presented a new input-based obfuscation technique called conditional code obfuscation that encodes or encrypts parts of the program using a key that is effectively removed from the program. The contributions for this work are:

- We presented the principles of an automated obfuscation scheme that can conceal condition-dependent malicious behavior from existing and future input oblivious malware analyzers.

- We have developed a prototype compiler-level obfuscator for Linux and performed experimental evaluation on several existing real-world malware programs, showing that a large fraction of trigger based malicious behavior can be successfully hidden.

- We provided insight into the strengths and weaknesses of our obfuscation. We discuss how an attacker equipped with this knowledge can modify programs to

increase the strength of the scheme. We also discuss the possibility of brute-force attacks as a weakness of our obfuscation and provide insight into how to develop more capable malware analyzers that incorporate input domain knowledge.

We then focused on a powerful obfuscation technique that recent malware has adopted to impede both white-box and grey box analyzers. This obfuscation technique converts the binary x86 code of malware into a random bytecode language and uses emulation to execute the code on the real hardware. We presented a technique that automatically reverse engineers the emulator and defeats the obfuscation. The contributions are:

- We formulated the research problem of automatic reverse engineering of malware emulators. By analyzing an emulator's execution trace using a given emulation model on how a bytecode instruction is fetched and executed, we can identify the bytecode region and discover the syntax and semantics of the bytecode instructions.

- We developed a framework and working prototype system that includes: a novel method to identify candidate memory regions containing bytecode, a method for identifying dispatch and instruction execution blocks, and a method for discovering bytecode instruction syntax and semantics. The output of our system can be used by existing analysis tools to analyze and extract malware behavior; for example, the identified bytecode can be converted to x86 instructions for static and/or dynamic analysis.

Finally, the dissertation presents a method to make host monitoring efficent and robust at the same time. This approach, which we call Secure In-VM Monitoring, has promise in protecting antivirus programs from unknown malware that may have attempted to disable the antivirus. By utilizing this technique, any malware residing

in the host cannot modify the data or hooks belonging to the antivirus. Therefore, even if an unknown malware resides in the system, it cannot disable the antivirus to allow other detectable malware to continue their attack. The contributions on making host monitoring more efficient and secure are:

- Leveraging hardware virtualization and memory protection features, we proposed the Secure In-VM Monitoring framework where a security monitor or antivirus tool can reside inside a guest VM but still enjoy the same security benefits of out-of-VM monitors (Section 7.3 and Section 7.4), bringing both efficiency and robustness.

- We have implemented a prototype of the SIM framework based on KVM and Windows guest OS (see Section 7.5). For demonstration, we have developed two security monitoring applications using our framework. We provide experimental evaluation of the performance overhead. (see Section 7.6).

- We provide a systematic security analysis of SIM against a number of possible threats throughout the paper, and show that SIM provides no less security guarantees than what can be achieved by out-of-VM monitors.

8.2 Future Work

While a number of problems in the malware analysis and system monitoring area have been addressed in this thesis, the research work has revealed several other areas to explore and a few open problems that may be worth solving to reach the goal of improving the effectiveness of our defense against malware. These problems and areas are described below:

- **Decompiling Emulated Malware:** We have presented in the thesis, methods to automatically reverse engineer emulators present in malware, and determine the syntax and semantics of the bytecode language instructions. However, this

is the first necessary step towards analyzing the malware in a useful manner. For example, in order to identify or extract signatures that can be used to detect this malware, parts of the malware may need to be decompiled to extract code that represents the malware's behavior. Our extracted syntax and semantics both provide the necessary basis to perform this decompilation. However, the issue arises with the exploration of different paths in the code. While we can identify the control-flow instructions, our dynamic analysis method extracts only one path for execution. It will be required to combine a multipath exploration technique that can solve the constraints and depend on the the control-flow semantics we extracted to find where another unexplored path starts from. In any case, the code for a single execution path might also be sufficient for a signature. These problems and issues need to be further investigated and explored.

- **Fine-grained Monitoring of The Host:** While our Secure In-VM Monitoring approach has provided a means to monitor the host at a finer grain than that which is possible using an out-of-VM approach, it is still prone to switching overhead that results from the pagetable transfers and TLB flushes. This kind of overhead is tolerable for monitoring events such as sytem calls or kernel functions. However, it is not sufficient to monitor a system at the granularity of control-flow or data-flow. The need for monitoring at that granularity has been shown by many systems that identify illegal control-flow[14] or data modifications [53, 16]. Although finding attacks at on the basis of control-flow and data-flow throughout the whole system is challenging, there are certain areas in the OS that might have specific properties which may be checked exclusively by a secure antivirus to verify whether it is behaving as it should. For example, the process scheduler, interrupt dispatcher, etc., may all be verified for integrity. Monitring at that level of granularity maybe possible by combining

run-time instrumentation using dynamic translation techniques and SIM.

- **Monitoring Kernel Module Behavior:** While processes have a very well-defined interface to the operating system that cannot be circumvented by the process alone, kernel modules can deliberately bypass the interface and directly modify data or execute code from anywhere else. This freedom makes it hard to monitor the behavior of a kernel module. Rootkits generally modify sensitive kernel data structures in order to hide malicious activities or change the operating system behavior. These events might be at the level of instructions. By enabling the monitoring of the kernel modules, antivirus programs can get more visibility of various actions of kernel modules in order to detect malicious ones. The challenge for solving this problem is to find an approach that can restrict kernel module behavior, so that legal behavior executes without much overhead and malicious behavior is intercepted.

8.3 Closing Remarks

This thesis addresses several research problems in the area of malware analysis and system monitoring. The problems that were targeted are mostly ones whose solutions can impact a large number of approaches of today and that may appear in the future. We targeted improvements in the area of both static malware analysis and dynamic malware analysis, and the obfuscations that we anticipated and defended against can affect large classes of analyzers. The system monitoring approach that we presented also can be directly used by antivirus programs today in order to improve their resilience against unknown malware in the target host and in effect improve the overall detection ability of the tools. The problems that we tackled in both malware analysis and system monitoring should provide benefits that can improve the effectiveness of malware detection systems employed in the industry. More efficient and robust malware analysis can lead to more accurate and comprehensive information extracted

from malware to improve the formation of signatures and reduce the time required to generate them. Moreover, robust and efficient system monitoring can enable the use of these signatures in an effective manner on the end hosts. Research on the open problems along this line can keep on improving the effectiveness of malware detection as a whole and let us reach closer and closer to our goal of defeating malware and gaining an upperhand over them once and for all.

REFERENCES

[1] "Anubis: Analyzing Unknown Binaries." http://anubis.seclab.tuwien.ac.at.

[2] "Cyveillance testing finds av vendors detect on average less than 19% of malware attacks." `http://www.cyveillance.com/web/news/press_rel/2010/2010-08-04.asp`. Last accessed Sep. 16, 2010.

[3] "DYNINST API." http://www.dyninst.org.

[4] "FileMon for Windows." http://technet.microsoft .com/en-us/sysinternals/bb896642.aspx.

[5] "IDA Pro Dissasember ." `http://www.datarescue.com/ida.htm`. Last accessed Sep. 16, 2010.

[6] "Ollydbg." `http://www.ollydbg.de`. Last accessed Sep. 16, 2010.

[7] "RegMon for Windows." http://technet.microsoft .com/en-us/sysinternals/bb896652.aspx.

[8] "Siemens stuxnet attack sophisticated, targeted." `http://www.controlglobal.com/industrynews/2010/163.html`. Last accessed Sep. 16, 2010.

[9] "VirtualPC." http://www.microsoft.com/windows/ products/winfamily/virtualpc/.

[10] "VMWare." http://www.vmware.com.

[11] "Norman Sandbox Whitepaper." http://www.norman .com/documents/wp_sandbox.pdf, 2003.

[12] "2007 malware report: The economic impact of viruses, spyware, adware, botnets, and other malicious code," 2007.

[13] "Annual report pandalabs 2008," 2008.

[14] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J., "Control-flow integrity," in *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*, 2005.

[15] Aho, A., Lam, M., Sethi, R., and Ullman, J., *Compilers—Principles, Techniques, & Tools*. Addison Wesley, 2006.

[16] Akritidis, P., Cadar, C., Raiciu, C., Costa, M., and Castro, M., "Preventing memory error exploits with wit," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[17] "Avira Antivirus." `http://www.free-av.com`. Last accessed Mar. 6, 2009.

[18] Bacher, P., Holz, T., Kotter, M., and Wicherski, G., "Know your enemy: Tracking botnets." http://www.honeynet.org/papers/bots, 2005.

[19] Bailey, M., Cooke, E., Jahanian, F., Watson, D., and Nazario, J., "The blaster worm: Then and now," *IEEE Security and Privacy*, vol. 3, pp. 26–31, 2005.

[20] Bailey, M., Oberheide, J., Andersen, J., Mao, Z. M., Jahanian, F., and Nazario, J., "Automated Classification and Analysis of Internet Malware," in *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, September 2007.

[21] Balakrishnan, G. and Reps, T., "Analyzing memory accesses in x86 executables," in *Proceedings of Compiler Construction (LNCS 2985)*, pp. 5–23, Springer Verlag, Apr. 2004.

[22] Balakrishnan, G. and Reps, T., "Recovery of variables and heap structure in x86 executables," Tech. Rep. 1533, Computer Sciences Department, University of Wisconsin–Madison, 2005.

[23] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," in *Proceedings of the Symposium on Operating System Principles*, October 2003.

[24] Bayer, U., Kruegel, C., and Kirdai, E., "TTanalyze: A Tool for Analyzing Malware," in *Proceedings of the 15th Annual Conference European Institute for Computer Antivirus Research (EICAR)*, pp. 180–192, 2006.

[25] Bell, J. R., "Threaded code," *Communications of the ACM*, vol. 16, June 1973.

[26] Bellard, F., "QEMU, a Fast and Portable Dynamic Translator," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 41–41, USENIX Association, 2005.

[27] Bishop, M., *Computer Security: Art and Science*. Addison-Wesley Professional, 2003.

[28] "BitBlaze." `http://bitblaze.cs.berkeley.edu`. Last accessed Sep. 16, 2010.

[29] Borders, K., Zhao, X., and Prakash, A., "Siren: Catching Evasive Malware (Short Paper)," *sp*, vol. 0, pp. 78–85, 2006.

[30] Breslau, L., Chase, C., Duffield, N., Fenner, B., Mao, Y., and Sen, S., "Vmscope: a virtual multicast vpn performance monitor," in *Proceedings of the 2006 SIGCOMM workshop on Internet network management*, 2006.

[31] Brumley, D., Hartwig, C., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., Song, D., and Yin, H., "Bitscope: Automatically dissecting malicious binaries," in *CMU-CS-07-133*, 2007.

[32] Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., and Yin, H., "Towards automatically identifying triggerbased behavior in malware using symbolic execution and binary analysis," *Technical Report CMU-CS-07-105, Carnegie Mellon University*, 2007.

[33] C. Collberg and C. Thomborson and D. Low, "A taxonomy of obfuscating transformations," in *Technical Report 148, The University of Auckland*, July 1997.

[34] C. Collberg and C. Thomborson and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL98)*, January 1998.

[35] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2003.

[36] C. Willems, "CWSandbox: Automatic Behaviour Analysis of Malware." `http://www.cwsandbox.org/`, 2006. Last accessed Sep. 16, 2010.

[37] Caballero, J., Yin, H., Liang, Z., and Song, D., "Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis," in *Proceedings of ACM Conference on Computer and Communication Security*, Oct. 2007.

[38] Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., and Engleri, D., "EXE: Automatically generating inputs of death," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2006.

[39] Cerven, P., *Crackproof Your Software: Protect Your Software Against Crackers*. No Starch Press, 2002.

[40] Christodorescu, M. and Jha, S., "Static analysis of executables to detect malicious patterns," in *Proceedings of the Usenix Security*, 2003.

[41] Christodorescu, M., Jha, S., Seshia, S., Song, D., and Bryant, R., "Semantics-aware malware detection," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.

[42] Christodorescu, M., Jha, S., Seshia, S. A., Song, D., and Bryant, R. E., "Semantics-Aware Malware Detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005)*, (Oakland, CA, USA), pp. 32–46, ACM Press, May 2005.

[43] Christodorescu, M., Kruegel, C., and Jha, S., "Mining Specifications of Malicious Behavior," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*, (New York, NY, USA), pp. 5–14, ACM Press, 2007.

[44] Cook, J. J., "Reverse execution of Java bytecode," *The Computer Journal*, vol. 45, no. 6, 2002.

[45] Crandall, J., Wassermann, G., Oliveira, D., Su, Z., Wu, F., and Chong, F., "Temporal search: Detecting hidden malware timebombs with virtualmachines," in *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[46] Cui, W., Peinado, M., Chen, K., Wang, H., and Irun-Briz, L., "Tupni: Automatic reverse engineering of input formats," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2008.

[47] da Wang, C. and Ju, S., "The dilemma of covert channels searching," in *ICISC*, pp. 169–174, 2005.

[48] Dagon, D., "Botnet detection and response: The network is the infection," in *OARC Workshop*, 2005.

[49] Data Rescue, "IDA Pro Disassembler and Debugger." `http://www.datarescue.com/idabase/index.htm`. Last accessed Sep. 16, 2010.

[50] Debaere, E. H. and Campenhout, J. M. V., *Interpretation and Instruction Path Coprocessing*. Cambridge, MA: MIT Press, 1990.

[51] Dinaburg, A., Royal, P., Sharif, M., and Lee, W., "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[52] Egele, M., Kruegel, C., Kirda, E., and Yin, H., "Dynamic spyware analysis," in *Proceedings of the 2007 Usenix Annual Conference (Usenix07)*, 2007.

[53] Erlingsson, U., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C., "Xfi: software guards for system address spaces," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[54] Feng, H., Giffin, J., Huang, Y., Jha, S., Lee, W., and Miller, B., "Formalizing sensitivity in static analysis for intrusion detection," in *Proceedings of the IEEE Symposium on Security and Privacy*, (Oakland, California), May 2004.

[55] Ferrante, J., Ottenstein, K., and Warren, J. D., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, July 1987.

[56] Ferrie, P., "Attacks on Virtual Machine Emulators." Symantec Advanced Threat Research, 2006.

[57] Ferrie, P., "Attacks on More Virtual Machines." http://pferrie.tripod.com/papers/attacks2.pdf, 2007.

[58] Filiol, E., "Strong cryptography armoured computer viruses forbidding code analysis," in *Proceedings of the Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2005.

[59] Futoransky, A., Kargieman, E., Sarraute, C., and Waissbein, A., "Foundations and applications for secure triggers," *ACM Transactions of Information Systems Security (TISSEC)*, vol. 9, Feb. 2006.

[60] Gao, D., Reiter, M. K., and Song, D., "On gray-box program tracking for anomaly detection," in *USENIX Security Symposium*, (San Diego, California), Aug. 2004.

[61] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D., "Terra: A virtual machine-based platform for trusted computing," in *Proceedings of ACM Symposium on Operating Systems Principles*, 2003.

[62] Garfinkel, T. and Rosenblum, M., "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2003.

[63] Garvey, T. and Lunt, T., "Model-based intrusion detection," in *Proceedings of the 14th National Computer Security Conf. (NCSC)*, (Baltimore, Maryland), June 1991.

[64] Ghosh, A., Schwartzbard, A., and Schatz, M., "Learning program behavior profiles for intrusion detection," in *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, (Santa Clara, California), April 1999.

[65] Giffin, J., Jha, S., and Miller, B., "Detecting manipulated remote call streams," in *Proceedings of the 11th USENIX Security Symposium*, (San Francisco, California), August 2002.

[66] Giffin, J. T., Jha, S., and Miller, B. P., "Automated discovery of mimicry attacks," in *9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, (Hamburg, Germany), Sept. 2006.

[67] Gryaznov, D., "An analysis of Cheeba," in *Proceedings of the Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 1992.

[68] Hoglund, G., *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[69] Hunt, G. and Brubacher, D., "Detours: Binary Interception of Win32 Functions," in *WINSYM'99: Proceedings of the 3rd conference on USENIX Windows NT Symposium*, (Berkeley, CA, USA), pp. 14–14, USENIX Association, 1999.

[70] Intel, *IA-32 Intel Architecture Software Developer's Manual Volume 3B: System Programming Guide, Part 1*, January 2006. Order Number: 253668-018.

[71] Jiang, X., Xu, D., and Wang, X., "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[72] Jiang, X., Wang, X., and Xu, D., "Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction," in *CCS '07: Proceedings of the 14th ACM conference on Computer and Communications Security*, (New York, NY, USA), pp. 128–138, ACM, 2007.

[73] Jiang, X., Xu, D., Wang, H. J., and Spafford, E. H., "Virtual Playgrounds for Worm Behavior Investigation," in *RAID*, pp. 1–21, 2005.

[74] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Antfarm: Tracking processes in a virtual machine environment," in *Proceedings of the USENIX Annual Technical Conference*, 2006.

[75] Kang, M. G., Yin, H., Hanna, S., McCamant, S., and Song, D., "Emulating emulation-resistant malware," in *Proceedings of the Workshop on Virtual Machine Security (VMSec)*, 2009.

[76] Kang, M. G., Poosankam, P., and Yin, H., "Renovo: a hidden code extractor for packed executables," in *Proceedings of WORM 2007*, 2007.

[77] Kang, M. G., Poosankam, P., and Yin, H., "Renovo: A Hidden Code Extractor for Packed Executables," in *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM'7)*, October 2007.

[78] Kelem, N. L. and Feiertag, R. J., "A separation model for virtual machine monitors," in *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 1991.

[79] Kirda, E., Kruegel, C., Banks, G., Vigna, G., and Kemmerer, R., "Behavior-based spyware detection," in *Proceedings of the USENIX Security Symposium*, 2006.

[80] Klint, P., "Interpretation techniques," *Software Pratice and Experience*, vol. 11, Sept. 1981.

[81] Ko, C., Fink, G., and Levitt, K., "Automated detection of vulnerabilities in privileged programs by execution monitoring," in *Proceedings of the 10th Annual Computer Security Applications Conference (ACSAC)*, (Orlando, Florida), December 1994.

[82] Kourai, K. and Chiba, S., "Hyperspector: Virtual distributed monitoring environments for secure intrusion detection," in *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, 2005.

[83] Kruegel, C., Kirda, E., Mutz, D., Robertson, W., and Vigna, G., "Automating mimicry attacks using static binary analysis," in *Proceedings of the USENIX Security Symposium*, (Baltimore], Maryland), August 2005.

[84] Kruegel, C., Robertson, W., and Vigna, G., "Detecting kernel-level rootkits through binary analysis," in *Proceedings of ACSAC*, 2004.

[85] Kruegel, C., Robertson, W., and Vigna, G., "Detecting Kernel-Level Rootkits Through Binary Analysis," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, (Tucson, AZ), pp. 91–100, December 2004.

[86] "Kernel based virtual machine." `http://www.linux-kvm.org/page/Main_Page`. Last accessed Sep. 16, 2010.

[87] Lanzi, A., Sharif, M., , and Lee, W., "K-tracer: A system for extracting kernel malware behavior," in *Proceedings of The 16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, 2009.

[88] Lattner, C. and Adve, V., "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[89] Lee, W., Stolfo, S., and Mok, K., "A data mining framework for building intrusion detection models," in *Proceedings of the IEEE Symposium on Security and Privacy*, (Oakland, California), May 1999.

[90] Lin, Z., Xu, D., and Zhang, X., "Automatic protocol format reverse engineering through context-aware monitored execution," in *Proceeding of the Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[91] Litty, L., Lagar-Cavilla, H. A., and Lie, D., "Hypervisor support for identifying covertly executing binaries," in *Proceedings of the 17th USENIX Security Symposium*, 2008.

[92] Madnick, S. E. and Donovan, J. J., "Application and analysis of the virtual machine approach to information system security and isolation," in *Proceedings of theWorkshop on Virtual Computer Systems*, 1973.

[93] Magnusson, P. S. and Samuelsson, D., "A compact intermediate format for SimICS," tech. rep., Swedish Institute of Computer Science, 1994.

[94] "Malfease Malware Repository." `https://malfease.oarci.net`. Last accessed Sep. 16, 2010.

[95] Martignoni, L., Paleari, R., Roglia, G. F., and Bruschi, D., "Testing cpu emulators," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[96] Martignoni, L., Christodorescu, M., and Jha, S., "Omniunpack: Fast, generic, and safe unpacking of malware," in *Proceedings of ACSAC*, 2007.

[97] Martignoni, L., Christodorescu, M., and Jha, S., "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware," in *ACSAC*, pp. 431–441, 2007.

[98] Moore, D., Shannon, C., , and Brown, J., "Code-red: A case study on the spread and victims of an internet worm," in *Proceedings of the ACM Internet Measurement Workshop*, 2002.

[99] Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., and Weaver, N., "Inside the slammer worm," *IEEE Security and Privacy*, vol. 1, no. 4, pp. 33–39, 2003.

[100] Moser, A., Kruegel, C., and Kirda, E., "Exploring multiple execution paths for malware analysis," in *Proceedings of the IEEE Symposium of Security and Privacy*, 2007.

[101] Moser, A., Kruegel, C., and Kirda, E., "Exploring multiple execution paths for malware analysis," in *Proceedings of the IEEE Symposium of Security and Privacy*, 2007.

[102] Moser, A., Kruegel, C., and Kirda, E., "Limits of static analysis for malware detection," in *Proceedings of ACSAC*, 2007.

[103] Newsome, J. and Song, D., "Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software," in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[104] Nielson, F., Nielson, H. R., and Hankin, C., *Principles of Program Analysis*. Springer-Verlag, 1999.

[105] "Offensive Computing." `https://www.offensivecomputing.net`. Last accessed Sep. 16, 2010.

[106] "Oreans Technologies." `http://www.oreans.com/`. Last accessed Mar. 6, 2009.

[107] "Oreans Technologies: Code Virtualizer." `http://www.oreans.com/codevirtualizer.php`. Last accessed Mar. 6, 2009.

[108] "Oreans Technologies: Themida." `http://www.oreans.com/`. Last accessed Mar. 6, 2009.

[109] Packet Storm Security. `http://www.packetstormsecurity.org/`. Last accessed Sep. 16, 2010.

[110] Payne, B., Carbone, M., Sharif, M., and Lee, W., "Lares: An architecture for secure active monitoring using virtualization," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[111] Payne, B. D., Carbone, M., and Lee, W., "Secure and flexible monitoring of virtual machines," in *Proceedings of the 23rd Annual Computer Security Applications Conference*, pp. 385 – 397, December 2007.

[112] "Futuremark PCMark 05." `http://www.futuremark.com/products/pcmark05/`. Last accessed Apr. 20, 2009.

[113] Petroni, N. L. and Fraser, T., "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *Proceedings of the 15th USENIX Security Symposium*, 2006.

[114] Petroni, N. L., Jr., T. F., Molina, J., and Arbaugh, W. A., "Copilot - a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[115] Petroni, N. L. and Hicks, M., "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the ACM conference on Computer and Communications Security*, 2007.

[116] Popov, I. V., Debray, S. K., and Andrews, G. R., "Binary obfuscation using signals," in *Proceedings of the USENIX Security Symposium*, 2007.

[117] Provos, N. and Holz, T., *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Reading: Addison-Wesley Professional, 2007.

[118] Raffetseder, T., Kruegel, C., and Kirda, E., "Detecting system emulators," in *ISC*, pp. 1–18, 2007.

[119] Riley, R., Jiang, X., and Xu, D., "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.

[120] Riordan, S. and Schneier, B., "Environmental key generation towards clueless agents," in *Mobile Agents and Security*, 1998.

[121] Roberts, P. F., "Targeted malware attacks: The new normal." `http://www.infoworld.com/t/hacking/targeted-malware-attacks-the-new-normal-159?page=0,1`. Last accessed Sep. 16, 2010.

[122] Royal, P., Halpin, M., Dagon, D., Edmonds, R., and Lee, W., "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *Proceedings of ACSAC*, 2006.

[123] Royal, P., Halpin, M., Dagon, D., Edmonds, R., and Lee, W., "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," in *ACSAC*, pp. 289–300, 2006.

[124] Rutkowska, J., "System virginity verifier." `http://www.invisiblethings.org/papers/hitb05_virginity_verifier.ppt`. Last accessed Sep. 16, 2010.

[125] Sekar, R., Bendre, M., Bollineni, P., and Dhurjati, D., "A fast automaton-based method for detecting anomalous program behaviors," in *Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California*, May 2001.

[126] Seshadri, A., Luk, M., Qu, N., and Perrig, A., "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proceedings of the ACM Symposium on Operating System Principles*, 2007.

[127] Sharif, M., Lanzi, A., Giffin, J., and Lee, W., "Impeding malware analysis using conditional code obfuscation," in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[128] Sharif, M., Lanzi, A., Giffin, J., and Lee, W., "Automatic reverse engineering of malware emulators," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[129] Sharif, M., Lanzi, A., Giffin, J., and Lee, W., "Rotalumé: A tool for automatically reverse engineering malware emulators," Tech. Rep. GT-CS-09-05, School of Computer Science, Georgia Institute of Technology, 2009. `http://www.cc.gatech.edu/research/reports/GT-CS-09-05.pdf`.

[130] Sharif, M., Lee, W., Cui, W., and Lanzi, A., "Secure In-VM Monitoring Using Hardware Virtualization," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2009.

[131] Sharif, M., Yegneswaran, V., Saidi, H., Porras, P., and Lee, W., "Eureka: A framework for enabling static malware analysis," in *Proceedings of the 13th European Symposium on Research in Computer Security (ESORICS)*, 2008.

[132] Silicon Realms, "Armadillo/software passport professional." `http://siliconrealms.com/`. Last accessed Mar. 6, 2009.

[133] Sinha, S., Harrold, M. J., and Rothermel, G., "Interprocedural control dependence," *ACM Transactions on Software Engineering and Methodology*, vol. 10, Apr. 2001.

[134] Sipser, M., *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[135] Sites, R. L., Chernoff, A., Kerk, M. B., Marks, M. P., and Robinson, S. G., "Binary translation," *Communication of the ACM*, vol. 36, Feb. 1993.

[136] Skulason, F., *1260-The Variable Virus*. Virus Bulletin, 1990.

[137] Smith, J. E. and Nair, R., *Virtual Machines: Versatile platforms for systems and processes*. Morgan Kaufmann, 2005.

[138] Stata, R. and Abadi, M., "A type system for Java bytecode subroutines," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, Jan. 1999.

[139] Stephen Pearce, "Viral polymorphism." VX Heavens, 2003.

[140] Stolfo, S. J., Wang, K., and Li, W.-J., "Fileprint analysis for malware detection," in *ACM CCS WORM*, 2005.

[141] Symantec - Virus Database, "Keylogger.stawin." `http://www.symantec.com/security_response/writeup.jsp?docid=2004-012915-2315-99`. Last accessed Sep. 16, 2010.

[142] Symantec - Virus Database, "Linux.slapper.worm." `http://securityresponse.symantec.com/avcenter/security/Content/2002.09.13.html`. Last accessed Sep. 16, 2010.

[143] Szor, P., *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[144] Tan, K., Killourhy, K. S., and Maxion, R. A., "Undermining an anomaly-based intrusion detection system using common exploits," in *Recent Advances in Intrusion Detection (RAID)*, (Zurich, Switzerland), Oct. 2002.

[145] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C., Anderson, A. V., Bennett, S. M., Kgi, A., Leung, F. H., and Smith, L., "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[146] Vasudevan, A. and Yerraballi, R., "Cobra: Fine-grained malware analysis using stealth localized-executions," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[147] Vasudevan, A. and Yerraballi, R., "Stealth Breakpoints," in *Proceedings of the 21st Annual Computer Security Applications Conference*, (Washington, DC, USA), pp. 381–392, IEEE Computer Society, 2005.

[148] Veldman, F., "Heuristic anti-virus technology." `http://vx.netlux.org/lib/static/vdat/epheurs1.htm`. Last accessed Sep. 16, 2010.

[149] "Virus Total." `http://www.virustotal.com/`. Last accessed Mar. 6, 2009.

[150] "VMPsoft VMProtect." `http://www.vmprotect.ru/`. Last accessed Mar. 6, 2009.

[151] Wagner, D., *Static Analysis and Computer Security: New Techniques for Software Assurance*. Ph.D. dissertation, University of California at Berkeley, 2000.

[152] Wagner, D. and Soto, P., "Mimicry attacks on host based intrusion detection systems," in *Proceedings of the Ninth ACM Conference on Computer and Communications Security (CCS)*, (Washington, DC), November 2002.

[153] Wahbe, R., Lucco, S., Anderson, T., and Graham, S., "Interprocedural control dependence," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, pp. 203–216, 1993.

[154] Wang, K., Parekh, J. J., and Stolfo, S. J., "Anagram: A content anomaly detector resistant to mimicry attack," in *Proceedings of RAID*, 2006.

[155] Wang, K. and Stolfo, S., "Anomalous payload-based network intrusion detection," in *Proceedings of RAID*, 2004.

[156] Wang, Y.-M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., and King, S. T., "Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities," in *NDSS*, 2006.

[157] Wang, Z., Jiang, X., Cui, W., and Wang, X., "Countering persistent kernel rootkits through systematic hook discovery," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.

[158] Wang, Z., Jiang, X., Cui, W., and Ning, P., "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[159] Wikipedia, "CIH (Computer Virus)." http://en.wikipedia.org/wiki/CIH_(computer_virus). Last accessed Sep. 16, 2010.

[160] Wikipedia, "Dr. Solomon's Antivirus Toolkit." http://en.wikipedia.org/wiki/Dr_Solomon's_Antivirus. Last accessed Sep. 16, 2010.

[161] Wikipedia, "Timeline of computer viruses and worms." http://en.wikipedia.org/wiki/Timeline_of_computer_viruses_and_worms. Last accessed Sep. 16, 2010.

[162] Wilhelm, J. and cker Chiueh, T., "A forced sampled execution approach to kernel rootkit identification," in *Proceedings of RAID*, 2007.

[163] Willems, C., Holz, T., and Freiling, F., "Toward Automated Dynamic Malware Analysis Using CWSandbox," *IEEE Security and Privacy*, vol. 5, no. 2, pp. 32–39, 2007.

[164] Wondracek, G., Kruegel, C., and Kirda, E., "Automatic network protocol analysis," in *Proceeding of the Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[165] Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G., "Linux security modules: General security support for the linux kernel," in *Proceedings of the 11th USENIX Security Symposium*, 2002.

[166] Wroblewski, G., "Generalmethod of program code obfuscation." PhD thesis, Wroclaw University of Technology, 2002.

[167] Yin, H., Liang, Z., and Song, D., "Hookfinder: Identifying and understanding malware hooking behaviors," in *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, 2008.

[168] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E., "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of ACM Conference on Computer and Communication Security*, Oct. 2007.

[169] Young, A. and Yung, M., "Cryptovirology: Extortion based security threats and countermeasures," in *Proceedings of the IEEE Symposium of Security and Privacy*, 1996.

[170] Zorz, Z., "Operation aurora malware investigated." http://www.net-security.org/malware_news.php?id=1223. Last accessed Sep. 16, 2010.